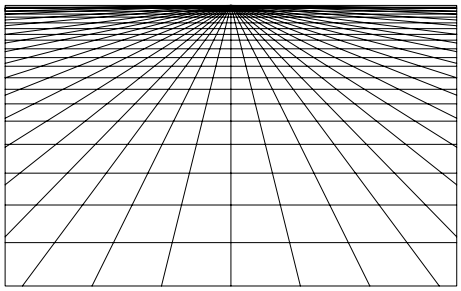
UNIVERSITY OF OSLO

FACULTY OF SOCIAL SCIENCES

TIK

Centre for technology, innovation and culture
P.O. BOX 1108 Blindern
N-0317 OSLO
Norway
http://www.tik.uio.no

Universiteit Maastricht



ESST

The European Inter-University Association on Society, Science and Technology
http://www.esst.uio.no

The ESST MA

Pleasure and Pain: Drift and Vulnerability in Software Systems

Christian V. Lundestad
University of Oslo / Universiteit Maastricht
Technological Culture
2003

Word Count: 24.740

Synopsis

Modern, Western post-industrial societies and our complex technological systems are subject to risks unparalleled in the history of mankind. These risks expose vulnerabilities in our technologies, our societies and our personal selves, as we become immersed in technologies without which our cultures cannot function. The proliferation of information and communication technologies (ICTs) into all aspects of life poses unique risks for all of us.

At the heart of ICTs lies the software which gives the computer its purpose. The aim of this thesis is to investigate how social and organizational factors influence the vulnerability of software systems and their users. The site where software is produced is studied through interviews among software developers. An interdisciplinary approach is employed; using theories of risk and vulnerability of complex technological systems, as well as theories from organizational sociology and software engineering. Scott A. Snook's theory of *practical drift* is used as the basis for further analysis.

Four areas are identified where social factors compel software developers to drift away from a global set of rules constituting software development processes and methods. Issues of *pleasure and control,* difference in *mental models*, undue *production pressures,* and *fragmentation of responsibility* all contribute to an uncoupling from established practices designed to guarantee the reliability of software.

The implications of these factors in terms of vulnerabilities of software systems, its users, and ultimately of our societies are discussed. Directions for future research are identified, and a hope for the future is expressed, where software will be produced that instead of *avoiding* risks, tries to *anticipate* them.

Keywords: risk, vulnerability, software vulnerability, practical drift, information society

Christian V. Lundestad
(christian@lundestad.com)
Oslo, October 6, 2003

The ESST M.A.
Specialization: Technological Culture
1st semester university: University of Oslo
2nd semester university: Universiteit Maastricht
Supervisor: Dr Anique Hommels, Universiteit Maastricht

Preface

Pleasure in the job puts perfection in the work.
-Aristotle

The title of this thesis hints at the emotions experienced by software developers during the course of a development project. They find immense pleasure in manipulating the most complex technological artefact known to mankind, creating complex structures and edifices from intangible materials, using not much more than the power of their minds. Their pleasure is only matched by the pain felt when things do not go as envisioned; when the software does not do what it is supposed to; or when the product of their labour does not meet with user approval.

For me, the process of writing this thesis has been exclusively a pleasurable experience, and I sincerely hope that Aristotle's maxim will hold true for the end product. For someone originally trained as a computer scientist and software engineer, it has been most rewarding to learn to see science and technology in a new light, and be able to do fieldwork among old colleagues.

I would like to thank the people at Telenor Mobile and FIRM who took time out from their busy schedules to be interviewed, especially Rodin Lie and Peter Myklebust who did most of the work recruiting other interviewees and acted as my "gate openers." I am particularly grateful to my supervisor, Dr Anique Hommels , for her invaluable assistance and insightful comments at the various stages of the work with this thesis. My fellow ESST students in Oslo and Maastricht also deserve many thanks for their friendship and inspiring discussions both on- and off-topic; especially my Maastricht flatmates Jo Anders Heir, Stian Slotterøy Johnsen, and Zeynep Bağcı.

Table of Contents

1 Vulnerable Software – Vulnerable Lives

1.1 Introduction

It is seven in the morning on Monday, October 26, 1992.[1] At the London Ambulance Service (LAS), the brand new, custom-built Computer Aided Despatch (CAD) system goes live. The control room is filled with excitement, but also apprehension. LAS is the world's largest ambulance service, and the staff are about to start using a computer system that is more advanced and complex than any ambulance service has ever had before. The CAD system is supposed to aid emergence despatch operators by automating many of the tasks associated with taking emergency calls from the public and despatching ambulances to the correct location. With the old system they had to take down details of an emergency on a piece of paper and put this note on a conveyor belt that would take it on to further processing. The new system has advanced features such as a computerised map system with public call box identification, vehicle location tracking, automatic update of resource availability, automatic identification of duplicate calls, and automatic ambulance mobilization in simple cases. Compared to the manual system they have been using until now, the CAD system represents a quantum leap into the future.

As the London morning rush gets underway it becomes clear that things are going terribly wrong. Some emergency calls appear to get "lost" in the system so no ambulance is sent to people in critical condition. The delays cause distressed people to call the emergency number again and again. This increase in the number of calls causes waiting times of up to 30 minutes before emergency calls can be dealt with. Other parts of the system fail too. The computerised map system refuses to recognize certain roads, forcing the operators to use

---

[1] The following narrative is based on Flowers, 1996, chap. 4, except where otherwise referenced.

maps and telephones to give directions to ambulance drivers. The automatic allocation of ambulances to accident sites forces emergency crews further and further away from their home bases and into unfamiliar parts of the city, adding further to the delays and confusion. Even more distress calls are generated when ambulances fail to turn up at accident sites, arrive late, or turn up two at a time.

In the confusion, ambulance crews, who have received little training in the new system, fail to operate it in its intended manner, leading the CAD system to base its directives on incorrect information. The crews in turn become increasingly frustrated with the CAD system, ignoring its orders, causing even more confusion, frustration, and delays, which further compound the problems in a deadly vicious circle. Within 36 hours of operation, the system has to be shut down, and the operators are forced to use a hybrid solution, allocating ambulances manually. One week later, the whole system locks up altogether, and LAS has to revert back to the fully manual paper-based system.

A later inquiry report (Page et al., 1993) concluded that "the computer system itself did not fail in a technical sense. Response times did on occasion become unacceptable, but overall the system did what it had been designed to do. However, much of the design had fatal flaws that would, and did, cumulatively lead to all symptoms of systems failure" (para. 1007x). Although no deaths have been directly linked to this incident (para. 6090-6091), it is clear that the introduction of the CAD system constituted an unacceptable risk to the inhabitants of London, jeopardizing their lives and health. The spectacular failure of the CAD system illustrates three kinds of vulnerabilities: the vulnerability of our personal selves, the vulnerability of our modern societies and the vulnerability of our computer systems. These vulnerabilities are the focus of this thesis.

1.2 The Role of Social Factors in Software Vulnerability

I have chosen the LAS case as the starting point of this thesis since it provides a poignant example of how computers in general and software in particular can affect our lives and our well-being. It is also a good and well-documented example of how it is not always the technical aspects of technologies that constitute the largest risk, but that social and organizational factors must be taken into account when attempting to understand the technologies we surround ourselves with, the risks they pose for us, and the vulnerabilities they expose both in us and themselves. In this context, I take vulnerability to mean the ability of people or systems to survive and continue to function when subjected to unwanted events. This notion of vulnerability is further elaborated in section 2.2.

Dealing with information systems and information networks has become part of daily life in our modern technological cultures. Our economies are fuelled by the exchange of information. International financial markets and the global news media are just a few examples of activities that would be severely affected by only minor disruptions in the services provided by the global information systems we have built. These information systems are made up of interconnected computers. In the last decade, the Internet has grown from being a tool for scholars and researchers communicating mainly via e-mail, to a global marketplace and an indispensable source of information. Businesses rely on the Internet for communication with their customers and partners and for financial transactions; private citizens use it for news, e-commerce, public information and keeping in touch with friends and family. We all rely on the continuing operation of the payroll systems of our employers; the reservation systems of airlines and railway companies; and the tax systems of our governments. In short, the Internet and other complex information systems are becoming part of the technological infrastructure that is enmeshed with our culture. It is therefore important

to understand the vulnerabilities of complex computer systems and come to terms with issues related to security, privacy and reliability.

Information systems are vulnerable in the sense that they are subject to hardware failure (e.g. disk crashes, power failures, component malfunctions), software failure ("bugs", logical errors, etc.), unauthorised access, deliberate attempts to disrupt operation ("cracker" attacks, software viruses, denial of service attacks), etc. Issues of privacy and confidentiality are aspects that become increasingly important as individuals, businesses, and governments use computer systems for storing and communicating information. In addition, computer systems may not be adequately able to handle unforeseen events or accommodate changes in the system's environment. These aspects will be discussed in detail later in this thesis.

The goal of my research has been to investigate some of the social and organizational factors that contribute to the vulnerability of computer systems. I have looked specifically at the development of complex software systems by software professionals and how social issues relating to their backgrounds, their values and ideals, their interaction with managers and clients, and their work settings can impact the vulnerability of the software they produce. By using theories of risk and vulnerability of complex socio-technical systems to investigate how social factors can lead to a deterioration of such systems, I hop to shed new light on important issues that so far have been overlooked when analysing the vulnerabilities of computer systems. Particular emphasis has been placed on Snook's (2000) theory of *practical drift*, especially the notion that local, pragmatic action within an organization can have detrimental and potentially catastrophic effects for the organization as a whole. Applying this theory to organizations constructing computer software and identifying factors that can contribute to this drift has been the main objective of this thesis. My research question can be summed up as: *How do social and organizational factors during design and development of software influence the vulnerability of software systems and their users?*

We will return to the London Ambulance Service case at the end of this thesis. Before that we will examine what characterizes software as a technological artefact and look at issues of risk and vulnerability surrounding the development and use of software. We will then meet some software professionals at two companies that have to deal with these issues on a daily basis. Hopefully, we will emerge at the end with new insights into some of the factors that contribute to the vulnerability of software systems and thereby the vulnerability of our modern cultures that have come to rely so heavily on these technologies.

1.3 Studying Social Aspects of Software Development and Use

By emphasising the social aspects of the design and production of technology, I place myself firmly in the tradition of researchers within the field of science, technology, and society (STS) studies. Rejecting a naïve technological determinism viewing technological innovation as external to society, STS scholars investigate how social factors influence the development of technologies, as well as the construction of meaning surrounding these artefacts. While still acknowledging that science and technology can bring about changes in society and social life, they point out that existing human relationships, established meaning formations, everyday practices, interactions, and social structures shape technological changes. By opening up the "black box" of technology, i.e. looking at the content of technology and how it is shaped in an interaction with society and culture, STS researchers hope to gain a deeper understanding of our technological cultures and broaden the technology policy agenda.[2] Researchers studying the social construction of technology emphasize the *malleability* of technologies, giving individuals and relevant social groups an *interpretative flexibility* in determining the social and cultural meanings of technological artefacts (Bijker, 1995).

---

[2] See Williams & Edge, 1996, for an excellent overview of the field. Although they prefer to call the field "social shaping of technology" (SST), I feel that the term STS is more in line with contemporary usage.

An interdisciplinary approach is the hallmark of STS studies. By drawing on research from philosophy, sociology, economy, and innovation studies, STS researchers use a diverse set of theories and methods in the study of science and technology, asking what is unique about science and technology in culture (Bowden, 1995, p. 77). Most STS scholars have concentrated on traditional, physical artefacts like bridges, domestic appliances, and rifles (Mackenzie & Wajcman, 1995); or bicycles and light bulbs (Bijker, 1995). The subject matter of this thesis is computer systems, or more specifically, software. In the following I will discuss what characterizes software and touch briefly on some of the STS theories that are relevant in understanding the social and cultural issues surrounding the development and use of software.

Computer systems are commonly viewed in terms of *hardware* and *software*. The hardware is the physical artefacts involved, with the digital computer and its silicon-based integrated circuits containing millions of transistors per square centimetre at its heart, providing a universal calculating engine. In some sense, the digital computer is the ultimate malleable artefact, having the ability to be changed to provide any functionality that involves the storage, manipulation and calculation of any data that can be represented in numerical (digital) form. Software is what makes this possible; providing the reconfiguration of the computer and the algorithmic programs that harness its generic properties into a specific application. The production of software is therefore interesting from an STS perspective, since software is constantly shaping and reshaping the computer, inventing and reinventing its social meaning, and reconfiguring human interactions around it. The massive interconnection of computers into global information networks that has taken place in the last decade is impressive in terms of the hardware involved, but equally so for the innovations in software which lie behind it, enabling computers to “talk” and in doing so opening new social world for us human beings.

The advanced technological artefacts surrounding us are also increasingly relying on software for their operation. Almost everything from our washing machines to our television sets and our automobiles depend on computers and the software that makes them "run". The fact that software is ubiquitous underlines the importance of investigating its characteristics and implications for our societies.

The phrase commonly used to describe modern, networked computer technologies is *information and communication technologies* (ICTs). ICTs have attracted increased interest from researchers of many fields as the application of these technologies has become commonplace both at home and at work. The increased ubiquity of ICTs in modern societies has had profound consequences and reshaped social and organizational activity, especially in the workplace. Previous STS-related research has uncovered the processes that shaped computers and information networks during the cold war (e.g. Abbate, 1999; Ceruzzi, 1999; Edwards, 1995; Edwards, 1996), stressing the point that the structures and configurations of current ICTs are the result of social and economic processes. When it comes to the social shaping of software, Williams and Edge (1996, pp. 882-884) group the research into three major strands: the organizational sociology of software, the "social constructivist" analysis of software, and studies of the commodification of software.

The organizational sociology of software has focused on studying the production and use of software, using theories and tools from industrial and organizational sociology. Researchers within this field have studied the division of labour and expertise during development of software, as well as the way gender and class relationships are changed or maintained through the application of computer systems in the workplace (e.g. Green et al., 1993), and how conflicts over control of ICTs emerge (e.g. Murray & Knights, 1990).

Social constructivist scholars have advocated the study of the scientific laboratory in order to investigate the construction of meaning in science and technology (Latour &

Woolgar, 1979/1986). The software development process can also be viewed as a kind of laboratory, and it has consequently proven to be an attractive site for researchers within this field. MacKenzie (1993) has criticized the attempt to use formal mathematical methods to improve the reliability of software on the grounds that mathematical "proofs" are not absolute and open for controversy. Low and Woolgar (1993) have studied how the classifications of certain issues within a software development project as technical "is a thoroughly social accomplishment" (p. 54). The efforts to create "artificial intelligence" or "knowledge-based systems" have also attracted attention from philosophers and sociologists of knowledge, who criticize computer scientists naïve hopes of replicating competent, socialized human action in a machine (e.g. Collins, 1995). Sally Wyatt (1998) has investigated the introduction of computer networks in government administration in the U.K. and the U.S., criticizing the technological determinism inherent in the belief that the mere establishment of a technical infrastructure would cause changes in work practices and social interactions.

Studies of the commodification of software have investigated how software has moved from bespoke applications tailored for a specific use to standardized "shrink-wrapped" packages. A case in point is the near monopoly of Microsoft's *Office* package, which dominates the market in business and home "productivity" applications such as word processing and electronic spreadsheets. The dynamics behind the stabilization and commodification of such applications have been investigated with a view to the economic and organizational implications of these developments (e.g. Brady et al., 1992).

Theories and methods from the social sciences have in the past decade attracted interest from researchers within *software engineering* and related fields. Unlike pure computer science theory, which usually only deals with technical issues intrinsic to the formal world of algorithms, data structures, and programming languages, researchers within software engineering deal with processes and methods for organizing software work and

communicating with future and existing users of software. Other software-related fields with a strong social element include *human-computer interaction* (HCI) and *computer-supported cooperative work* (CSCW). Scholars within these fields have turned to the social sciences in an attempt to find methods and tools that can help them improve their understanding of the social issues involved in software development, especially issues surrounding understanding the work done in the social settings in which the software is to be deployed and eliciting requirements from customers, users and other stakeholders in the software development process.[3] The emphasis in these fields is usually not on investigating and understanding current software development practices as might be the case in a more traditional social science approach. It is rather about prescribing changes to existing practices in an attempt to intervene and improve them (Rönkkö & Lindeberg, 2000). Ethnography and other methods from social science then run the risk of being just another tool in the software developers' toolbox, subordinate to the perhaps overly positivist mode of thinking within this profession. Floyd et al. (1992) provide a pioneering attempt to investigate the epistemological and philosophical foundations underlying software development. By bringing together scholars from philosophy, social science, informatics, and mathematics they seek to investigate computer scientists' and software engineers' notions of truth and proofs, their use of metaphors, and their view of their own reality and that of their users. By doing so, they are able to shed new light on old problems within the field of software development. Dittrich et al. (2002) is the most recent contribution of this interdisciplinary research into the conflation of *social thinking* and *software practice*. Their goal is to "promote the discourse about the interrelationship of social science-based approaches that shed light on the social aspects of software practice" (Klischewski et al., 2002, p. ix), by deconstructing software practice and related research, questioning established paradigms, analysing how social aspects of software

[3] See Quintas (1993) for a collection of articles on the subject.

use are conceptualized, promoting a broader understanding of the software development process, adapting social thinking for improving software development methods, relating software practice to organizational change, and reorienting software practice by focusing on use-oriented design (pp. x-xi). This interdisciplinary approach and pioneering research has provided important inspiration for this thesis.

1.4 Bridging Social Thinking and Software Practice

The groundbreaking studies of the social processes surrounding the production and use of technological artefacts proved to be a breakthrough in understanding how technological artefacts are shaped by social activities, and how the social activities themselves are shaped by these artefacts. As we saw above, computers and software have been a ripe arena for STS research. This has been important research, expanding our knowledge about the role of computers and software in society. It has made us aware of the fact that ICTs can be used for different purposes, some more appealing in the eyes of different groups than others. Bringing different stakeholders and interest groups together to influence the development of ICTs can only be to the benefit of all involved.

A more limited amount of research has gone into studying the software developers and their social world. This is perhaps due to the difficulty of doing fieldwork among software developers, who mostly sit at a computer programming, typing in what is aptly named “code”. The intricacies of software development are usually poorly understood by non-professionals, making ethnographic studies of software professionals a frustrating task for social scientists; Low and Woolgar (1993) found the “technical talk” among software developers who were the target of their ethnographic fieldwork to be of “complete unintelligibility” (p.50). Within their field, software engineering researchers have embraced theories and methods from social science in order to improve their processes and to better understand the users, thus improving

the software they produce. Except for some pioneering, interdisciplinary minded scholars, however, they have generally not sought to use these tools in an attempt to investigate the epistemological and philosophical foundations of their own field.

This "gap" between software engineering researchers trained in engineering and natural sciences, and philosophers and social science researchers has meant that the positivist assumptions behind much software engineering theory and methods have gone relatively unchallenged. This makes the construction of software a ripe arena for STS researchers, who with their interdisciplinary approach should be able to bring new insights to the field. This is what I hope to accomplish with this thesis. I intend to draw from the STS field a constructivist approach to understanding the design and production of technological artefacts. In addition, I will draw important ideas and inspiration from some of the more STS-minded studies of software professionals. Theories from organizational sociology on risk and accidents combined with recent research on software development will provide the analytical framework. Drawing empirical data from two case studies and using my personal knowledge and experience from the software engineering field, I hope to contribute new insights into how social factors influence how software is built and used, and how this affects the vulnerability of both the software and those who use it.

1.5 Method

Due to the scope of this thesis and the time and resource constraints involved, it was necessary to limit the original research question to something manageable, finding a path of inquiry that was feasible and that seemed likely to yield interesting results. By using some of the most influential theories of risk and vulnerabilities of complex technical systems (Perrow, 1999/1984; Snook, 2000) as a theoretical foundation, it was possible to limit the scope of the original research question and formulate issues for further empirical work.

The main empirical basis for this thesis is a series of twelve interviews with software professionals in two companies. All the interviewees are directly involved with the development of software, either as developers (programmers), software architects, or managers. The main goal of these interviews was to identify social factors in everyday software development work that could have an impact on the quality and thus the vulnerability of the software developed by the interviewees. The interviews took the form of semi-structured qualitative interviews, lasting approximately one hour each.

When determining which companies to contact in order to obtain interviewees, I approached personal acquaintances in the IT industry that were placed in such a way within their organization that they could act as "gate openers". By getting these key people interested and involved in my thesis, they could put me in touch with individuals within their organizations who would be sympathetic to my research and who would have the authority to allocate the time and resources needed to conduct the interviews. This is in accordance with Stake's criteria for case selection, "selecting a case of some typicality, but leaning towards those cases that seem to offer *opportunity to learn*" (1994, p. 243, original emphasis), and to select "that case from which we feel we can learn the most" (ibid.).

The two companies ultimately selected as interview sites also provided intriguing contrasts. Telenor Mobile is the largest mobile telecommunications operator in Norway. It has 1600 employees working with every aspect of mobile telecommunications. Software development is just one among a wide variety of activities. The software developers interviewed work in a section within the software development department responsible for the development and maintenance of "middleware" software – a highly technical domain without "end-users" in the traditional sense. The other company, FIRM, is a small entrepreneurial upstart with only 40 employees in Norway. The development of their Internet-based market-

research software product is their main activity. They produce highly visible software with a wide range of non-technical users. The two companies are further described in section 3.2.

My empirical research can be considered to be what Stake (1994) calls an "*instrumental case study,* [where] a particular case is examined in order to provide insight into an issue or refinement of theory. The case is of secondary interest; it plays a supportive role, facilitating our understanding of something else" (p. 237, original emphasis). The focus was therefore not on the specific conditions at Telenor Mobile and FIRM, but on what I could learn about software developers and their attitudes towards risks and vulnerabilities, and the social and organizational factors that influence their work. My goal was to solicit "insider accounts" from these developers. According to Hammersley and Atkinson, accounts are important "for what they may be able to tell us about those who produced them. We can use what people say as evidence about their perspectives" (1995, p.125).

Scholars have argued that interviews in the classical research tradition presuppose a particular epistemological position, assuming the existence of a social world "that is independent of the language used to describe it" (Seale, 1998, p. 202). The opposite position would be an *idealist* one "in which interview data – or indeed any account of the social world – are seen as presenting but one of many possible worlds" (p. 203). When analyzing the interview material, it was important to read the software developers accounts not as describing any objective social reality, but as their subjective perception of their world. Since no other groups at Telenor Mobile and FIRM were interviewed, no definitive claims can be made about the social configurations within these two companies. Nevertheless, since the present research investigates the attitudes of software developers and the social factors influencing them and their work, I believe the interview data constitute a valid basis for further analysis. Seale distinguishes between treating the interview as a *topic*, investigating how language is used in the interviews; or as a *resource,* gathering data about the external

world from interviewees' accounts (p. 204). Although the emphasis in my research is more on the actual accounts of the software developers, important data can be gleaned by investigating the language employed by the interviewees and their linguistic repertoires, for instance when talking about their pleasures in programming. In this sense, the interview material is treated as *both* topic *and* resource.

At Telenor Mobile, developers were asked to volunteer as interviewees. At FIRM, interviewees were selected by my "gatekeeper", the director of development. At both sites, the interviewees constituted a significant portion of the total number of developers, diminishing the potential for bias. Given the nature of the research, I believe that the selection of interviewees did not have significant impact on the data that was acquired through the interviews. The majority of the people interviewed at Telenor Mobile and FIRM were software developers and software architects who routinely do programming as part of their normal work. In addition, at Telenor Mobile the section manager directly in charge of the developers was interviewed; at FIRM both the director of development and the quality assurance (QA) manager were interviewed. A complete list of the interviewees can be found in Appendix A.

The interviewees were relatively homogeneous when it comes to age and education; they were all aged from the late twenties to the mid thirties, having Master-level educations from one of the four Norwegian universities. This is a common background for Norwegian software developers. In addition, all the interviewees were male, a fact that sadly reflects the current state of affairs in the Norwegian IT industry. In 1996, only 8 % of first year students at the computer science and communications technology studies at the

Norwegian University of Science and Technology[4] were female (*Women in Computing,* n.d., para. 1-2). With only 50 % of the female students completing the 4.5 years Master program, this means that female software developers are a rare sight in Norwegian companies. Although this historical low point sparked efforts to increase the number of women in these professions, the results of these efforts are yet to be seen in the workplace. I therefore hope that the reader will forgive me for using the male pronoun when referring to the singular software developer. While other researchers have looked explicitly at gender differences among computer scientists and software developers (e.g. Rasmussen & Håpnes, 1991; Kleif & Faulkner, 2003), this is not an issue in this thesis.

As recommended by Seale, a topic guide was prepared before the interviews, containing topics that were to be covered during the interview. Nevertheless, I attempted to be as *non-directive* as possible, asking open-ended questions and encouraging the interviewees to tell their story in their own words. Since I and all the interviewees are native Norwegians, the interviews were conducted in the Norwegian language. I do not believe that this had any undue influence on the outcome of the interviews. The terminology in the software development field is heavily influenced by English, and I believe that the main concepts of *risk* and *vulnerability* have Norwegian counterparts that have very close, if not identical, semantic contents. The interviews were recorded and later transcribed in order to facilitate further analysis. Direct quotes were translated into English by me before inclusion in this thesis. In doing so, I have attempted to strike a balance between following the original wording and conveying the tone of the original.

---

[4] The Norwegian University of Science and Technology (NTNU) in Trondheim is the main site for computer science education in Norway, having more Master-level computer science graduates then all the other colleges and universities combined.

To the extent that my own background as a software developer[5] influenced the interviewees and the material gathered, I believe this to have enabled me to establish a better rapport with the interviewees, showing them that I was familiar with their area of expertise, their language and terminology, as well as their norms and values. Any bias on my part would have to be blamed on the same familiarity and personal identification with the interviewees, perhaps contributing to a tendency to view matters from the point of view of the developers and a "blind spot" to different perspectives. On the other hand, Collins (1984) advocates "participant comprehension", demanding that "the investigator *him/herself* should come to be able to act in the same way as the native members 'as a matter of course'" (p. 61, original emphasis). In this sense, my past as a "native" software developer should have enabled me to achieve a much better comprehension of the software developers than most ethnographers venturing into this field.

In addition to my own empirical material, I studied other cases from the literature on risk, failures and accident involving software and computer systems (Leveson, 1995; Flowers, 1996; Neumann, 1995; *Library of failed information systems projects,* n. d.). Especially inspiring was the London Ambulance Service case (Page et al., 1993), which started this thesis. These cases provided a valuable background for my empirical work and poignant examples of the vulnerabilities of software systems, as well as the vulnerabilities of those who use them.

1.6 Structure of the Thesis

In chapter 2 I will present theories pertaining to risk and vulnerability. Ulrich Beck's notion of the *risk society* will be introduced, and I will discuss the idea that we also live an

---

[5] I hold a *sivilingeniør* (Master level) degree in computer science from the Norwegian University of Science and Technology (1995) and have worked as a software developer for more than six years. I was also an employee of FIRM for most of 1999.

information society and how this affects our vulnerabilities. Charles Perrow's *normal accident theory* and Scott A. Snook's theory of *practical drift* will be discussed in detail, since they form the basis for the subsequent analysis of the empirical material. Chapter 3 contains the results from interviewing 12 software professionals in two Norwegian companies about their attitudes towards risks and vulnerabilities in their daily work. Emphasising issues of pleasure and control, the formation of *mental models*, fragmentation of responsibility, and production pressures, I investigate whether Perrow and Snook's theories can be fruitfully applied to the development of software systems. Finally, in chapter 4 I summarize the findings, discuss their implications for software engineering work, and try to identify directions for further research. An appendix at the end lists all the software professionals interviewed during the work with this thesis.

2 The Risky Information Society

2.1 Introduction: The Risk Society

As citizens of modern, Western societies, we are surrounded by pervasive scientific, technological and industrial developments, without which modern society cannot be imagined. In short, we are immersed in *technological cultures* (Bijker, 2001). While providing us with an unparalleled standard of living, consumer products and inexpensive energy, the inescapable consequences of these developments are a set of risks and hazards that also are unparalleled in the history of mankind. These risks and hazards are no longer limited in time and space, and there is no one to be held accountable. Accidents in nuclear power plants such as Three Mile Island and Chernobyl can cause radioactive material to enter the atmosphere, making large areas far away from the plant itself uninhabitable and increasing the risk of cancer and foetal deformation for generations to come. Routine discharges of technetium-99 from the Sellafield reprocessing plant on the Western coast of England can be found in marine life as far away as the Svalbard islands in the Arctic (Martiniussen, 2002). The burning of oil and coal in power plants in the U.K. and Central Europe are known to cause acid rain in Scandinavia (*Acid Rain*, 2001).

The *Collins English Dictionary* defines risk as "the possibility of incurring misfortune or loss; hazard" (Hanks, 1986, p. 1318). A British Royal Society study group set up to investigate risks in engineering and public perceptions of risk, defines risk as "the probability that a particular adverse event occurs during a stated period of time, or results from a particular challenge" (Warner, 1992). According to Renn (1992), "the term *risk* denotes the possibility that an undesirable state of reality (adverse effects) may occur as a result of natural events or human activities" (p. 56, original emphasis). A high risk denotes a greater probability that adverse effects will occur than with a low risk. Renn notes that all concepts of

risk presuppose a distinction between reality and possibility, since within a fatalistic belief in a predetermined future the term risk makes no sense. A concept of risk implies making a causal connection between events and their effects, thereby making it possible to avoid or mitigate the adverse effects by avoiding or modifying the causal events.

Exactly which activities or phenomena constitute risks is open to dispute, since the concept of risk is "open to social definition and construction" (Beck, 1986/1992, p. 23). How risks are perceived is therefore an issue with profound social, economic and political implications. Any attempt to define a specific phenomenon as a risk and quantify the degree of risk involved is destined to be disputed by groups with conflicting interests. Nevertheless, the continuous assessment of risks and the weighing of risks versus perceived benefits seem to have become an integral part of life in our modern societies. The impossibility of determining any objective risks involved in human activities makes the *perception* of these risks much more important when investigating the influence of risks on people's behaviour and individual assessments of risks. The role of the media has therefore been the subject of research into the public perception of risk.

Risk scholars have in the last decade shifted our view of accidents and disasters from seeing them as the product of random, freakish events, to having social, organizational causes. The poison gas leak at Bhopal in 1984, the radiation leak from Chernobyl in 1986, and other fatal disasters spawned much research into the causes of such tremendous technical failures, shifting the focus from technical malfunctions to the social and organizational configurations that contributed to these accidents (Jasanoff, 1994).

Since the publication of Ulrich Beck's (1986/1992) seminal work, the notion of the *risk society* has become an important concept in both theoretical and political discourse. According to Beck, the industrial society that was the child of modernity is growing into the risk society of post-modernity. While modern, industrial society was about the distribution of

benefits ("goods") from industrial production, the risk society is about distribution of the risks ("bads") that are the inevitable, *complementary* consequences of the industrial society. These risks are not distributed equally alongside the benefits, but will often be imposed on people who are not in a position to benefit from the "goods" of this risk production.

The risks unique to the post-industrial society are related to our increased reliance on complex technological systems, without which we would be helpless. The electricity, gas, and water infrastructures; the road and rail networks; sea and air transport; when we use these technologies we accept that there are risks involved in using them. We subject ourselves to the risk of death or injury every day by using gas ovens, driving a car, or getting into an airplane. We accept those risks because we feel they are greatly outweighed by the benefits of these technologies. A different set of risks involved with the use of these technologies might be overlooked, however. The risks we run by making our societies rely on the uninterrupted operation of these technological systems, usually do not come to the forefront until an accident or failure makes them clear to us. Recent electricity blackouts in North America, Scandinavia and Italy are cases in point.

The terrorist attacks of recent years have focused our attention on the risks from external forces, such as the possibilities for malicious assaults on our technical infrastructures, cities and businesses. The September 11, 2001 attack on the World Trade Centre in New York showed us that our technological artefacts and complex systems can be turned against us, becoming the tools of terrorists. The reality and horror of these events and the external risks notwithstanding, we should not be distracted from other, internal risks associated with complex socio-technical systems. The ways we manage our technological systems and organize their operation may also be sources of risk. As we shall see later, the complexities involved in designing and operating modern technologies can themselves be sources of risk.

The last two decades have seen an explosive growth in a new form of complex technological systems. While information and communications technologies (ICTs) can trace their origins back 60 years, advances in ICTs have only recently made them ubiquitous in the work place and in our homes. Without modern computer and telecommunication networks working properly, global financial markets would collapse, national and international transport would grind to a standstill, and groceries would no longer fill the shelves of our supermarkets. Indeed, many commentators have claimed that since *information* is the primary commodity of modern society, we can speak of an *information society* qualitatively different from previous eras. An influential voice here has been the Spanish sociologist Manuel Castells with his three-volume work entitled *The Information Age* (Castells, 1996-8). However, as Frank Webster points out (Webster, 2002, p. 8), Castells and other researchers have greatly differing opinions as to what constitutes an information society and how to measure the degree to which a society can be said to be informational. While some theorists emphasise the emergence of technological artefacts, most notably information and communications technologies (ICTs), others see changes in economic, occupational, spatial or cultural configurations as more indicative of an information society. According to Webster, it is difficult to identify any quantitative or qualitative measures that unequivocally set the "information society" apart from previous eras.

Webster's critique notwithstanding, I think it would be rash to dismiss the information society altogether. I believe it is important to investigate why this notion has found such wide acceptance, especially among politicians. There is widespread use of the term in public and political discourse. Although theorists routinely employ the term in a broader sense, in general discourse the emerging information society seems to be closely associated with the growing number of interconnected computers in physical information networks (most notably the Internet) and the consequential fall in cross-border communication and organizational

costs. The term is mainly used to denote the possibilities and challenges posed by the growing use of ICTs and information networks in our increasingly globalised societies. It is this narrower meaning of the term "information society" that will be the basis for the present analysis.

In the information society, ICTs and the media play a major role in the formation of risks, risks sensibilities and risk perceptions (Van Loon, 2000). As Van Loon points out:

> As the global economy, the world political order and most socio-cultural systems are nowadays bound to high-speed and high-frequency information flows, there is no escape from the impact of telecommunications on processes of decision-making and anticipation. However, apart from accelerating information flows, ICTs also contribute to the acceleration of risks. (Van Loon, 2002, p. 12)

The mere speed of information exchange diminishes the time available for contemplation and reflection, with the consequences that may have for decision-making and political processes. Further investigation into the role of ICTs and risks in the information society is therefore warranted. The proliferation of ICTs also has consequences for the vulnerability of our societies, as we shall see in the next section.

2.2 Vulnerability of the Information Society

The *Collins English Dictionary* defines "vulnerable" as "1. capable of being physically or emotionally wounded or hurt. 2. open to temptation, persuasion, censure, etc." (Hanks, 1986, p. 1702). *Vulnerability* is then the quality or state of being vulnerable. Blaikie et al. (1994) see this term in the context of natural hazards:

> By vulnerability we mean the characteristics of a person or group in terms of their capacity to anticipate, cope with, resist, and recover from the impact of a natural hazard. It involves a combination of factors that determine the degree to which someone's life and livelihood is put at risk by a discrete and identifiable event in nature or in society. (p. 9).

These definitions emphasise the vulnerability of individuals or groups of individuals when faced with unwanted events. However, vulnerability can also be exhibited by systems. Einarsson and Rausand (1998) employ the term to "describe the properties of an industrial system that may weaken its ability to survive and perform in the presence of threats" (p. 535), focusing on "the (business) survivability of the system" (p. 542). Systems in this context may be "societies or states, a population of inhabitants in a certain geographical region, companies or technical systems" (Wackers, n. d., Various definitions). When investigating the vulnerability of software systems, the emphasis will be on this systemic definition of vulnerability, since:

> Vulnerability is also often used in relation to a society's information and communication infrastructures. A society's vulnerability to technical breakdowns, electronic terrorism and electronic warfare rises in accordance with the increasing centrality of ICT infrastructures for important sectors of society (finance, administration, defense, business). (Ibid.)

Clearly, the ubiquity of ICTs in the Western world has profound implications for our personal vulnerabilities and the vulnerability of our societies.

When discussing the hazards contributing to the vulnerability of ICTs rather then the abstract notion of vulnerability, I prefer to use the plural term *vulnerabilities,* denoting the diverse set of "weak spots" that make a system susceptible to damage or failure. Vulnerabilities can be exploited by *external* forces outside the system itself, or they can be the source of *internal* systemic failures. ICTs are clearly vulnerable to a long list of threats, both external and internal. Power failures and other physical hazards such as earthquakes, water

floods, etc. can seriously impair a systems ability to function as intended. These vulnerabilities can best be dealt with by physical safety measures such as redundant power supplies or duplicate systems in different locations. Other external threats may take advantage of weak points that are the result of internal vulnerabilities. Attempts at unauthorized access ("cracker" attacks[6]); deliberate attempts to disrupt the normal operation of systems through denial of service attacks[7]; computer viruses, worms, and "Trojan horses;"[8] all of these are external threats that can compromise computer systems. These malicious attacks are motivated by a variety of factors, but have become commonplace in today's interconnected world. They need to be taken very seriously by anyone with a computer connected to the Internet or any other computer network.

The internal vulnerabilities of a computer system are normally related to software. Flaws and errors in the program code are known as "bugs,"[9] and are inevitable in any program of non-trivial size. The exponential cost of tracking down and fixing bugs is a well known phenomenon for software professionals. Some bugs make the computer "crash" (i.e. cease normal program execution), while others cause more subtle errors in calculations or data manipulation. Others again may give rise to security flaws that can be exploited in the ways mentioned above.

---

[6] In popular usage, the term "hacker" denotes someone who attempts to gain unauthorized access to computer systems. Dedicated programming virtuosos feel that this is an unfortunate appropriation of a term they reserve for themselves. They prefer to call the computer criminals "crackers".

[7] For instance by flooding a system with requests for service, tying up computing resources and bandwidth to the detriment of legitimate users.

[8] A computer virus is a small program that spreads itself from computer to computer by attaching itself to other programs or files, possibly carrying a destructive "payload". A worm is similar to a virus, but does not need another program to spread itself. A Trojan horse is a malicious program masquerading as a useful program, but containing malicious software that for instance creates a "back door" into the affected computer to be exploited at a later time.

[9] The origin of the word is usually attributed to an episode in the early days of computing, when a computer malfunction was found to be caused by an insect ("bug") who had managed to get inside the computer, thereby causing the failure. Hence also the word "debugging" for the process of tracking down software errors.

Apart from the systemic vulnerabilities associated with ICTs, we also have to consider the vulnerabilities imposed by ICTs on their users, as well as society in general. As mentioned above, security flaws or software bugs can give unauthorized persons access to information systems, possibly violating the privacy or confidentiality of individuals or corporations. These vulnerabilities are becoming increasingly important as companies and governments use computer systems to store and process information about their customers or citizens. The existence of false or misleading information can have grave consequences for individuals, for instance in connection with credit ratings or criminal records.

Another aspect of the vulnerability of the users of ICTs relates to the extent that computer systems expose their users to hazards. People regularly trust their lives and safety to computers; modern aircraft are for instance “fly by wire”, i.e. fully under the control of computers; computers monitor nuclear power plants and control medical equipment. Fatal accidents that were traced to software errors have occurred in all these areas. Other computer systems may not put their users’ life in direct jeopardy, but still put them at risk when their functions and capabilities do not match the requirements of the users. A computer system could be operating without technical flaws, but still fail to provide its users with the functionality they need in order to perform their tasks. The system could also be inadequately equipped to handle unforeseen events or accommodate changes in its environment. The mismatch between user requirements and the actual capabilities of the computer system could compel users to change their interactions with the system, using it in ways not intended by its original designers. This could in turn expose further vulnerabilities both in the system itself and in its users. Vulnerabilities of this kind are predominantly related to the design of software, and therefore highly relevant to this thesis and the research question at hand. Examining the conditions under which software is developed, and learning as much as we can

about the vulnerabilities of computer systems, has therefore become crucial in assessing the vulnerability of the information society as a whole.

In order to establish a theoretical framework with which to analyse the social and organizational factors contributing to these vulnerabilities, we turn to two influential theories within the research on high-risk systems: Charles Perrow's *normal accident theory* and Scott A. Snook's theory on *practical drift.* These are covered in the next two sections.

2.3 Normal Accidents vs. High Reliability

One of the seminal works on risks and accidents is Perrow's *Normal accidents* (1984/1999). By examining a large number of accidents in fields as diverse as nuclear power plants, petrochemical plants, aircraft, the space program, and DNA research, Perrow is able to formulate *normal accident theory* (NAT). His basic tenet is that accidents are inherent to any technical or socio-technical system exhibiting certain characteristics; in such systems accidents are bound to happen – it is in this sense that accidents are *normal*.

In order to analyse a system's propensity for accidents, Perrow introduces the concepts of *complexity* and *coupling.* To measure a system's complexity we must look at the interactions between the components of the system. Most systems are designed with *linear* interactions in mind. Linear interactions are the well understood, sequential interactions where a component will typically get its input from an "upstream" component, do some sort of transformation, and subsequently deliver its output to a "downstream" component. If one component fails, it is relatively easy to locate and understand the point of failure and consequently handle it without catastrophic results.

On the other hand, if a component of the system serves multiple functions or is connected to several other components, the interactions are said to be *complex.* A component such as a water pump in a nuclear plant may be used for several different tasks, reducing the

costs of the plant. A failure of this one component, however, will affect the operation of the system in a much more serious manner than in the linear case. The failure can manifest itself in ways that make the source of the failure difficult to locate and handle, thus increasing the potential for disaster. The possibilities for unplanned and unexpected sequences of events are much greater in systems with complex interactions.

The concept of *coupling* is used by Perrow to classify systems according to the strength of the connections between their internal components. The term *tight coupling* is meant to describe a situation where there is no slack or buffer between two items, so that what happens in one directly affects what happens in the other. This originally mechanical term is used as a metaphor for systems where there are more time-dependent processes; more invariant sequences; there is only one way of reaching the goal; and there is little slack in supplies, equipment and personnel. Conversely, in *loosely coupled* systems processing delays are possible, the order of sequences can be changed, alternative methods to achieve the goal are possible, and slack in resources is possible.

Traditionally, accidents like the one at the Three Mile Island nuclear power plant in the U.S. in 1979 have been blamed on human error, specifically errors on the part of the operators responsible for monitoring and controlling the plant. Perrow's view is that the cause of such accidents must be sought after in other places than in the apparent failure of human operators. The sheer complexity of these large technical systems makes the interactions incomprehensible to any individual or even group of individuals. Accordingly, accidents are inherent properties of the complex and tightly coupled system.

It is important to emphasis that a system is not either complex or linear, nor is it either tightly or loosely coupled. Any system will have both complex and linear interactions and tightly coupled as well as loosely coupled subsystems. Perrow's point is that the more

complex interactions a system exhibits, and the more tightly coupled it is, the more the risk of accidents increases, and so the vulnerability of the system.

Since Perrow's theory sees accidents as "normal" within tightly coupled, complex systems, it has been criticized for not being able to prescribe remedies that could help us build safer systems. Indeed, Perrow's only solution seems to be that we abandon the idea of building complex systems like nuclear power plants altogether, because the risks are too great to bear. Perrow's theory also underplays the dynamic aspects of systems and organizations, since he does not discuss how the risks and vulnerabilities can change over time. For Perrow, these are *inherent properties* of the systems which can only be changed by changing the system itself to make it more loosely coupled and reducing the number of complex interactions.

To se the relevance of Perrow's theory for this investigation into the risks of software systems, we have to look at some of the unique properties of software that make it inherently complex. Every part of a software system is unique. Almost by definition, if a software developer has produced two equal modules of the same system, then he has not done his job properly, since the code for one module could have been reused in the other. In fact, much of a software developer's task consists of finding similarities and abstracting behaviour in such a way that the same code can be used in as many situations as possible. This means that any software system engineered after these principles will exhibit a very high degree of complexity, since any component of the system will be used for several different tasks and will be connected to many other subcomponents. A failure of any such component will therefore have a large detrimental impact on the operation of the system as a whole. An error in a subcomponent that is widely used across the system can be hard to track down and fix.

Coupling is a concept that is used in software engineering theory as well as in accident theory. A widely use software engineering textbook defines coupling as:

> Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point in which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" … caused when errors occur at one location and propagate through a system. (Pressman & Ince, 2000, p. 347)

Clearly, this notion of coupling is related to Perrow's concept, although the match is not complete. While software engineering practice dictates minimizing the degree of coupling by confining the interactions between components to a limited number of clearly defined interconnections, this does not necessarily decrease the coupling in Perrow's sense. The interconnections can still be strong even if they are limited in number. Processes can still be time-dependent; there can be invariant sequences; and little slack in the interactions between components, even in a system with low degrees of coupling in the software engineering sense of the word. In fact, all of these factors which constitute tight coupling in Perrow's sense are abundant in most software system. Normal accident theory therefore tells us that failures in software systems should come as no surprise, they are to be expected, and thus *normal*.

Perrow's theory has drawn criticism from the STS field, where scholars have pointed out that his position is a technological determinist one. By viewing accidents as purely causal effects of the properties of technological systems, he underestimates the influence of social factors on these issues, and prematurely absolves us from the responsibility for technological disasters. He also takes away our ability to deal with these issues through social and organizational measures. While still providing us with important tools for evaluating the risks and vulnerabilities associated with specific technological system, we need to elaborate further on his ideas in order to arrive at a satisfying theoretical basis for the present research questions.

Perrow and other proponents of normal accident theory study dramatic accidents and spectacular systemic failures, stressing the inevitability of accidents in complex systems.

Other researchers have concentrated on *successful* complex organizations, or *high reliability organizations* (Roberts, 1990; Sagan, 1993). High-reliability theorists tend to take a more optimistic view of our ability to manage complex systems and organizations. While still acknowledging failures as inevitable, they advocate technical and organizational measures such as continuous training, accountability, and redundancy (duplicating technical subsystems or organizational functions to reduce the likelihood of accidents should one component fail). These measures are believed to reduce the risks to acceptable levels.

For a more dynamic view of risks and vulnerabilities, combining the features of normal accident theory and high reliability theory, and providing us with a basis for analysing the social and organizational aspects of risks and vulnerabilities, we turn to Scott A. Snook and his theory of *practical drift.*

2.4 Practical Drift

In his book *Friendly fire* (2000), Scott A. Snook undertakes a thorough analysis of the 1994 incidents when two U.S. Air Force fighter planes accidentally shot down two U.S. Army helicopters carrying U.N. peacekeepers over Northern Iraq after erroneously identifying the friendly Black Hawk helicopters as enemy Iraqi Hinds. An Airborne Warning and Control System (AWACS) aircraft, equipped with highly advanced radar and communication equipment was in place, monitoring air traffic and communicating with aircraft in the area, but did not act to prevent the fighter planes engaging and shooting down the friendly helicopters. All 26 people aboard the helicopters perished. No serious technical malfunction could be found and no single human error could explain how this tragedy could occur. Consequently, the causes of the accident had to be sought in the social and organizational fabric of military operations. By investigating the incident and tracking the events that led up to the fatal shooting, Snook is able to formulate a theory of organizational breakdown that

offers an explanation of how a tragedy like the friendly fire incident over Northern Iraq can happen in an organization that is specifically designed to avoid this kind of accident. By developing accounts of the accident on several levels, from the individual, group, and organizational levels to a cross-level account, Snook builds his theory of *practical drift* – "the slow, steady uncoupling of practice from written procedure" (p. 194).

Snook argues that in any organization, rules and procedures laid down when designing the organization will not be able to cover all situations that may arise. Rules may even be conflicting, or minute adherence to detailed procedure may not be possible for practical reasons or may be in direct conflict with the overall task at hand. Also, the operational environment of the organization will change over time, rendering rules and procedures increasingly obsolete. In such situations, local practices will develop that are seen as *locally efficient*, growing out of the logic of the task in hand. Snook calls this "practical action." Across all levels of his analysis and with all actors involved, Snook finds that "globally untoward action is justified by locally acceptable procedure" (p. 182). Practical action is the driving force behind practical drift.

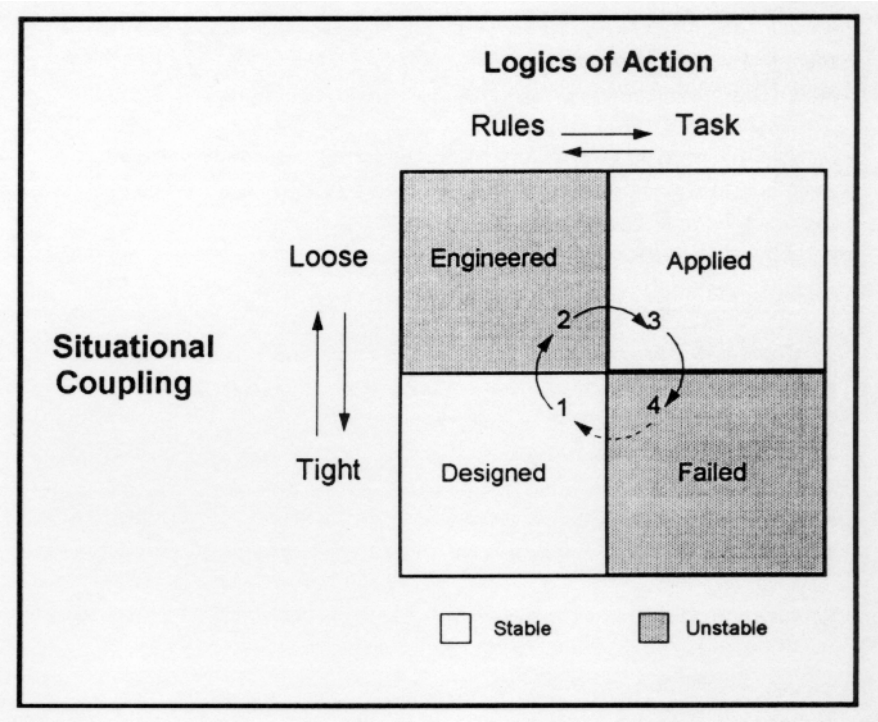


Figure 1: Theoretical Matrix (Adapted from Snook, 2000, p. 186)

The matrix in Figure 1 illustrates Snook's theory of practical drift. It captures three dimensions: *situational coupling, logics of action,* and *time.* The vertical axis represents the *degree of coupling* inside an organization. This concept is borrowed from Perrow's accident theory as described above, and refers to the level of interdependence between subunits within the organization. In a tightly coupled organization, each subunit has frequent communications and interactions with most of the other subunits and the actions of one subunit directly affect what happens in the other subunits. Unlike Perrow, however, Snook focuses on the dynamic nature of coupling. The patterns of interdependence between subunits change back and forth over time as the organization faces different contexts. Different situations encountered by the organization entail different demands in terms of subunit cooperation. A loosely coupled situational context can be handled by one or a low number of subunits without involving the other subunits, thereby minimizing the need for communication and interaction. In a tightly coupled situation, however, coordinated actions from many or all of the subunits are required.

In the military domain of Snook's research, the routine day-to-day air operations in the "no fly zone" in Northern Iraq constituted a loosely coupled situation, requiring little interaction between the army and air force subunits of the military task force. However, a combat situation requiring the joint efforts of all military branches is a very tightly coupled situation indeed, requiring communication and interaction between all subunits involved, and the observance of globally understood rules. As we shall see later, this transition from a loosely coupled to a tightly coupled situational context is where practical drift can manifest its most dangerous consequences.

The horizontal axis represents what Snook calls *logics of action,* defined as "systems of scripts, norms, and schemas among which people shift according to context" (p. 188). They are "context-dependent mind-sets or frames that influence behavior in predictable ways" (ibid.). Members of an organization change their logics of action depending on the context they are faced with. In this case between rule-based logics of action, where the rules take precedence in governing people's actions, and task-based logics of action, where behaviour is adapted to solving the tasks at hand.

The circle of arrows in the middle of the diagram represents time. The arrows suggest a circular motion between the four quadrants of the matrix; each quadrant representing a different combination of situational coupling and logics of action. Quadrant 1 is the system as designed on the drawing board, with a set of global rules and procedures prescribed by the designers. The rules of an organization will frequently be designed with the most demanding and tightly coupled situation in mind, making sure that such a situation will be handled correctly. Quadrant 2 represents an organization that is operating as designed. The transition from quadrant 1 to quadrant 2 happens when an organization is set up; in Snook's case when the United States launched "Operation Provide Comfort" in Northern Iraq and the "Combined Task Force Provide Comfort" took to the field. At first, the novice members of the

organization followed the rules as they were designed by the Pentagon system planners. The situation and organization were new to everybody involved, so rules were followed under the assumption that not following them could have severe consequences, at least in the form of punishment from superiors. The rules were designed with a tightly coupled, worst-case combat scenario in mind. This scenario would require extensive interactions between all the different military units involved in the operation. However, the actual, day-to-day operations of the task force did not require these extensive interactions. Instead, each unit found itself performing its tasks relatively independent of the others. U.S. Army helicopters would ferry military personnel and U.N. peacekeepers in and out of the area, while Air Force fighter planes would patrol the no fly zone. Airborne air traffic controllers would monitor the skies above Northern Iraq in their highly advanced AWACS "control tower in the sky". In effect, the situation was loosely coupled, but with the members of the organization following rules designed for a very tightly coupled situation. This is an unstable state of affairs, and the point where practical drift sets in.

The transition to quadrant 3 of the matrix is characterized by a shift from a rule-based to a task-based logics of action. As mentioned, the members of the organization at first feel obligated to follow the detailed set of rules set up to govern the general operation of the organization and the interactions between its subunits. However, as they become more familiar with the operational context, the focus shifts from following the rules to completing the tasks at hand. Since the organization encounters loosely coupled situations most of the time, the detailed rules designed to cope with tightly coupled situations may feel overly controlling and an unreasonable burden. The rules may not even match the practical situations that arise in real life operation; they may be difficult to comply with within the constraints of time and resources; or following the rules may actually be in direct conflict with the tasks at hand. Over time, global rules may become increasingly irrelevant to the local situations. The

completion of tasks takes precedence over following the rules, and locally practical interpretations and adaptations occur. As Snook puts it, "[w]hen the rules don't match, pragmatic individuals adjust their behavior accordingly; they act in ways that better align with their perceptions of current demands. In short, they break the rules" (p. 193). It is worth to point out that practical drift is not a symptom of wilfully negligent behaviour of members of the organization – this is normal. Not in Perrow's sense that it is a direct property of the system, but in the sense that this is normal behaviour by normal people. The process is usually gradual and imperceptible, hence the notion of *drift*. As Snook puts it: "After extended periods of time, locally *practical* actions within subgroups gradually *drift* away from originally established procedures" (p. 194, original emphasis). Usually, the demand for efficiency at the local level is the reason behind the drift and determines its path. Therefore, "[o]ver time, incremental actions in accordance with the drift meet no resistance, are implicitly reinforced, and hence become institutionally accepted within each subunit" (p. 194). Each day that passes without any adverse effects from the local adaptation of rules will grow a false sense of confidence and contribute to a reinforcement of the local practices.

Different subgroups will tend to develop different local adaptations to the globally defined rules. Different local contexts entail different pragmatics in response to the tasks of the subunit. In Task Force Provide Comfort, incoming aircrews were initially briefed on the colour schemes of enemy aircraft. For some reason, this requirement disappeared or "drifted away" over the years. Army helicopters were initially required to file detailed flight plans and stay in constant communication with AWACS crews. However, since helicopter flights were done on an ad-hoc basis and the destinations often were unknown at the start of the flight, it was not possible to complete flight plans in advance and keep AWACS controllers informed. Although people within one unit may have adapted rules and procedures to fit their local context, they will usually assume that people in the rest of the organization are sticking to the

original set of globally established rules. They are not aware of the drift that has occurred in other units, which is what can have disastrous consequences as the situational context changes.

When a loosely coupled situation suddenly becomes a tightly coupled one, we move from quadrant 3 to quadrant 4. For some reason, a coordinated effort of many or all subunits is required; making communication, interaction and adherence to common rules and procedures necessary. Years of drift away from the established baseline of global rules along different paths result in a catastrophic coordination failure as each unit follows its own locally adapted set of procedures, while at the same time expecting every other subgroup to be following the original set of baseline rules and procedures. In the skies above Northern Iraq, two fighter planes shot down two friendly helicopters while a crew of AWACS controllers looked on. On the other hand, the assumption that others will follow the formal rules is a rational one, and the best that any member of the organization can do.

The final part of the cycle in Figure 1, transitioning from quadrant 4 back to quadrant 1 represents a redesign of the organization to prevent the accident from ever happening again. However, the danger is that in an attempt to remedy every possible cause of the accident, the result is to overshoot, burdening the organization with even more complex rules and procedures, thus setting the stage for a new turn of the cycle. As Perrow puts it, "[t]he tighter the rules, the greater the potential for sizable practical drift to occur as the inevitable influence of local tasks takes hold" (p. 201).

The theories of Perrow and Snook provide an interesting framework with which to analyse the risks and vulnerabilities exhibited by software systems. We have seen that the complexity of software system is comparable to that of the industrial systems studied by Perrow. I intend to show that the dynamics of software development organizations can be compared to that of the military organizations studied by Snook. Armed with these theories,

we are ready to meet the software professionals producing complex software systems and investigate to what extent their organizations are subject to practical drift, what social factors contribute to this process, and whether this affects the vulnerability of the software.

3 Software Development: Pleasure or Pain?

3.1 Introduction

Considering the hyper-modern, high-tech world that software developers inhabit, it is perhaps surprising that personal skills play a crucial role in their work life. Developers judge themselves and others on the basis of craftsmanship and dedication to their job. Most of them will emphasize that computer programming is as much an art as a craft, involving high levels of creativity and the occasional stroke of genius. Ironically, while software developers produce tools that will make the work life of others more automated and less dependent on individual skills, their own work is still labour-intensive and dependant on the knowledge and skill of each developer. The software professional has much in common with the medieval stonemason, who used his superior skill to fashion great cathedrals out of stone. Software developers may work with more intangible materials, but like the great artisans of the Middle Ages, the software developers take great pride in their work, experiencing immense pleasure when they see great structures rising from humble building blocks; but also great pain when the results of painstaking work crumble before their eyes.

3.2 Two Companies – Two Contexts

In order to gather empirical data to form a basis for research into social and organizational factors influencing the work of software developers, I interviewed 12 software professionals in two companies based in Oslo, Norway. One, Telenor Mobile, is a large mobile telecommunications company where the software systems are an integral part of a larger operation. The other, FIRM, is a smaller company that markets and sells its own Internet-based market research software. These two companies are described in the following.

3.2.1 Telenor Mobile

Telenor Mobile[10] is the largest Norwegian mobile telecommunications operator. It is a subsidiary of Telenor, the successor to the state-owned telecommunications monopoly. Telenor is still the dominant telecommunications provider in Norway and is listed on the Oslo Stock Exchange, as well as the NASDAQ Stock Market in the U.S. Telenor Mobile has about 2.3 million customers[11], the vast majority subscribers to its GSM digital mobile telephone services, which have been in operation since 1994. Data about these customers are stored in several computer systems, ranging from mainframe systems to modern database systems. Some of these systems date back several decades while others are recent developments. The systems represent a wide range of hardware and software platforms.

The data stored about customers include information about phone calls that form the basis for billing the customers, as well as data pertaining to the telecommunication services and products that the customers subscribe to. These could be voice mail, short message services (SMS), WAP, e-mail, fax services, etc. Some data could be stored in several places at once, creating a potential for inconsistent data across different systems. These data need to be accessed in a variety of ways, by different organizational units within Telenor Mobile, by different computer systems and in different contexts. For instance, data about telephone usage need to be accessed by the billing systems in order to produce the quarterly telephone bill. The same data may have to be accessed by the customer service department when a customer calls in to complain about the size of the bill. When a customer wants to add a new service to her subscription, she could do this by calling customer service or do it through a web interface on the Internet.

---

[10] Web site: http://www.tmc.telenor.com
[11] Norway has approximately 4.5 million inhabitants, making more than half of the population Telenor Mobile customers.

The development of new products and services is seen as essential in order to maintain customer satisfaction and to keep Telenor Mobile's reputation as one of the world's most technologically advanced mobile telecom providers. This means that new products and services are constantly being developed in order to provide the customers with what are termed "value-added services", i.e. services that add value to the customers' mobile subscriptions beyond basic voice communication.

In order to satisfy the needs of a range of different client applications to access data dispersed over a multitude of different computer systems, Telenor Mobile has established a so-called *middleware* layer within its computer system architecture. The middleware layer forms a well-known interface for client applications (collectively known as *channels*), hiding the complexities of accessing customer data from the so-called backend systems. By abstracting the process of communicating with the backend systems, channels can be developed without detailed knowledge of the inner workings of the middleware layer and the backend systems. Conversely, the backend systems can be developed and maintained without specific knowledge about the channels. From the client applications' view, the middleware layer and backend systems form a "black box" that is understood only by its external interface defined by its input and output data. The responsibility for developing this middleware is in the hands of a dedicated section of the IT department of Telenor. This middleware section consists of around 20 project managers, software architects, software developers, and testers. In order to do their job, the architects and developers have to communicate with managers, marketing people, and other software professionals developing channels and backend systems. I conducted personal interviews with the head of this department and 7 of the developers who are responsible for developing and maintaining the middleware.

3.2.2 FIRM

Future Information Research Management (FIRM)[12] was founded in 1996. Bjørn Haugland, the founder of the company, had previously founded one of the most successful market research companies in Norway. He felt that the emerging Internet technology could be used to revolutionize the market research industry by automating a lot of the manual work involved in putting together questionnaires, collecting responses, tabulating data, and producing reports. Response data could instead be easily collected by using World Wide Web technologies and Internet questionnaires. FIRM's main software product, *ConfirmIT*, allows customers to produce online surveys using a 100 % web-based application. Application areas for the product include market research studies, customer feedback and employee satisfaction surveys. Feedback data is immediately accessible through an online reporting tool; a huge improvement from the weeks needed to code and tabulate data with traditional methods. FIRM's clients include some of the largest market research companies in the world, as well as a number of global corporations. FIRM is currently a privately held company with offices in Oslo, Stockholm, London, New York and San Francisco.

Working for a relatively new, upstart company, FIRM's developers have the luxury of being able to employ the latest technologies at their disposal. They do not have to deal with a legacy of old systems and technologies, but can gain a competitive advantage by using "cutting edge" tools and systems. This fact is actively used both in marketing and recruiting, and attracts developers with a desire to work with the latest "toys." Development of the ConfirmIT product is done in Oslo by a team of 14 software developers, 5 of whom were interviewed during the work with this thesis.

---

[12] Web site: http://www.firmglobal.com

3.2.3 Contexts for Software Development

The two companies, Telenor Mobile and FIRM, provide interesting contrasts as well as similarities. The developers at each of the companies are remarkably similar in terms of age and background and they all have software development as their primary job tasks. The contexts in which they perform these tasks differ, however.

The most conspicuous difference between Telenor Mobile and FIRM is in their size. Telenor Mobile with its 1600 employees is a large company by Norwegian standards, and is in turn part of the Telenor Group, a multinational corporation with interests in 16 countries and more than 14,000 employees in Norway alone. Telenor Mobile's activities span a wide range of different products and services associated with providing mobile telephony, e.g. operating the network infrastructure and marketing mobile services. Software development is just a small part of the activities within Telenor Mobile, and the middleware department the interviewees were drawn from, is a small section within the larger ICT staff. Consequently, the developers have several layers of management to deal with, and will rarely have any contact with the top executives of the company.

FIRM, on the other hand, is a much smaller company with about 40 employees in Oslo and about 60 worldwide. It is centred on the development, marketing and sale of the ConfirmIT software product, making software development its core activity. Although FIRM has offices in several countries, the bulk of the software development is done by the 14 member development team in Oslo. Since the development of software is FIRM's *raison d'être*, developers enjoy a relatively high status within the company. The whole of the Oslo office is located on one floor, making the work environment intimate and informal.

Another important area where the two companies differ is their relationship with the users of their software. The middleware software developed at Telenor Mobile does not have end-users in the normal sense; the "users" of the software are other computer programs that

treat the middleware as “black-box” functionality providing basic services that can be combined to provide higher order services for end-users such as mobile telephony customers. The software developed by the developers interviewed at Telenor Mobile does not have the kind of user interface we normally associate with computer programs, where human operators can interact with the systems using a screen and keyboard. This does not mean that the middleware developers do not have to relate to any users, however. There are still “customers” that request functionality and from whom requirements have to be elicited.

The developers at FIRM have a different relationship to their users since they develop an Internet-based application that will be used by human end-users. A complex graphical user interface (GUI) forms the boundary between the computer system and the person operating it. The developers will therefore have to take into account the fact that the systems will be used by people with varying degrees of computer skills. Some of the end-users are known at the time of development, but most of them will use the system after it is developed and put into production. The end-users will be highly skilled in the application domain of the application, in this case market research, most probably surpassing the software developers’ insights into the same area.

These companies provide interesting case studies, as they afford us a glimpse into the world of the professional software developers. By interviewing people heavily involved in the practice of creating software, we can gain insights into their social world and see which factors influence the vulnerabilities of the software they produce. Having developers from two different companies and two different contexts allows us to investigate whether these differences affect the vulnerabilities of their respective software products. Similarities can also be important, as they hint to important issues that can provide a basis for further research and theory building.

3.3 Practical Drift in Software Development Organizations

In order to apply Snook's theory of practical drift to software systems, we need to look at the conditions under which these are developed. Recall Figure 1 on page 33, which illustrates Snook's theory. The starting point is quadrant 1, with an organization that is created on the drawing board. Rules and regulations are defined to cope with the most tightly coupled situation imaginable and covering all situations foreseen by the designers. In a software engineering organization, we can compare this to the processes and organizational setup prescribed by a plan-oriented software process such as the Unified Software Development Process (Jacobson et al., 1999). The transition to quadrant 2 would be starting a software development project following the process to the letter. According to Snook's theory, this will be an unstable state of affairs since the rules are designed for a tightly coupled organizational set-up, but in most cases this will be "overkill" when it comes to the actual situation. We would expect the rule-oriented behaviour to be replaced by a task-oriented behaviour as individuals and subgroups within the organization pragmatically adjust their actions to be more in tune with the practical demands of the tasks at hand. These adjustments will in practice contribute to "practical drift" away from the baseline of rules and methods prescribed by the process. When attempting to establish whether practical drift occurs in software development organizations and the possible sources of such drift, we need to look at three areas. First, whether methods and processes in fact contribute to the overall quality of software, and thus a lower degree of vulnerability. Secondly, we need to establish which forces can contribute to practical drift during software development. Finally, we have to investigate how this can have a detrimental effect on the software produced.

3.4 Software Methods and Processes

Compared to other engineering discipline, the construction of software has a short history. Modern computers arose from code breaking efforts in the U.K. and U.S. during World War II. When computing was still in its infancy, the distinction between hardware and software was blurred. Programming the computer consisted of toggling switched on the computer's console or feeding it punched cards with instructions in binary machine code. In the late 1950s and early 1960s the first programming languages that were independent of a specific computer started to appear, and so software development became a separate task. Due to the still limited capacities of the hardware, computer programs were short, designed for specific tasks, and offered limited possibilities for human input. Each piece of software was developed for use in a specific location and usually developed by in-house programmers. Most software was "batch programs" which ran overnight without user intervention.

Fuelled by advances in solid-state physics, semiconductors, and electronics, the power and capabilities of digital computers increased exponentially in the post-war decades. As the computers grew smaller, faster, and more powerful, they got better facilities for user interaction and demanded new kinds of software. It soon became clear that the software could not keep up with the advances in hardware in terms of performance and reliability. Relative costs of software when compared to hardware soared; experienced software developers were hard to find; and the few who existed were not able to keep up with the demands for increasingly complex software as the computer constantly found new applications in government, the military, and private companies. The existing tools and programming languages proved to be inadequate for the new tasks that had to be solved if the new computers were to realise their full potential. Software development projects were more often than not fraught with enormous delays and cost overruns. The resulting software was often faulty, poorly documented and almost impossible to understand and maintain by others than

the original developers. This situation sparked an intense effort within research communities to come up with solutions to what was dubbed the "software crisis" by the end of the 1960s.

This marked the birth of the *software engineering* field. It was felt that the practice of software development should be modelled on other engineering disciplines, like civil engineering (i.e. building houses, bridges, roads, dams, etc.), mechanical engineering, and even computer hardware engineering. These practices had proven successful in taking activities that were once crafts based heavily on personal skills and turning them into predictable, scientifically based disciplines capable of forming the basis for industrial production. By dividing software development into independent tasks and prescribing systematic procedures for structuring and producing the programming code itself (*methods)*; for structuring the interaction between developers, managers, customers and users in predictable and quantifiable *processes*; as well as prescribing which *artefacts* (documents and program code) should be produced during the development process, it was believed that software development would become every bit as predictable and manageable as the other engineering disciplines. Consequently, a lot of effort has been put into creating methods and processes for software development in the last decades. The perhaps best known and most influential of the so-called plan-oriented approaches is the Unified Software Development Process (USDP[13]) (Jacobson et al., 1999). Other important software development processes include the Dynamic Systems Development Process (DSDM) (http://www.dsdm.com) and Extreme Programming (XP) (Newkirk & Martin, 2001).

Most of these processes rely on dividing the software development activities into *phases* with specific tasks, timeframes and milestone artefacts for each phase, modelled on the phases of for instance a bridge construction project. The classic phases of software

---

[13] Perhaps better known in its commercial incarnation as the Rational Unified Process (RUP), developed by the Rational Corporation, now a part of IBM. See http://www.rational.com for more information.

engineering are *analysis*, where feasibility studies are performed and requirements are gathered from users and other stakeholders; *design*, with detailed planning about how the system is going to be programmed; *construction*, where the actual programming and testing is performed and documentation is written; and *deployment*, where the finished system is installed for use in its target location. Along with *maintenance*, which covers systems management and service after deployment, these phases are referred to as a system's *lifecycle*.

Software methods and processes can be said to be *strict* or *rigid* if they contain detailed descriptions of the content of the phases, tasks and artefacts to be produced, prescribing checklists and procedures for error reporting, code review and other activities that are not directly related to programming. Such rigid processes purport to result in predictable software development projects by mimicking the processes that have proven effective in industrial production; in effect leaving little to the discretion of the individual developer. Many early processes prescribed strict separation of these phases, especially between the analysis and design phases on one hand and the construction phase on the other. It was felt that by having specially trained systems architects and designers analyse and divide the development tasks into independent subtasks that could be handled by individual programmers, previous problems with unstructured, unmaintainable code produced when programmers were left to their own devices could be avoided. This would in effect concentrate all the creative, analytical work to the analysis and design phases, leaving only menial, routine work to the programmers.

Neither Telenor Mobile nor FIRM can be said to employ rigid processes or methods. In fact, neither company uses any specific, named process, but both have developed their own processes based on ideas from the more influential methodologies mentioned above. The managers see it as important to have processes that are internalized and actually used by the developers. If the processes do not gain approval from the developers, they run the risk of

becoming “shelfware:” binders of irrelevant documents gathering dust on a shelf. Both companies have developed software tools that support their processes, containing flowcharts and checklists guiding the developers through each step in the process. The companies also constantly try to improve their processes through regular evaluations. It is also a goal to standardize the processes used; for Telenor Mobile across different departments and sections; for FIRM across offices in different countries. FIRM is perhaps the company which has seen the greatest change in attitude towards methods in recent years. Starting from scratch with only a handful of developer in 1996, the emphasis was more on creativity and speed in reaching the market with a new product, and less on establishing processes for software development. As the company grew and more developers got involved, it became clear that formal processes were needed to create a more predictable work atmosphere for the developers; to meet customers’ demands and expectations; and to increase the quality of the software itself. FIRM has therefore cooperated with software engineering and process quality researchers in order to develop a tailor-made development process. They now employ a flexible approach to software development. The overarching process can be tailored to the size and demands of each individual project, providing the developers with a range of processes to choose from. A simple change in some internal system functionality might be done “ad hoc”, i.e. without any formal measures; while development of a customized feature for a big client will be done with all the checks and safeguards in place.

The interviewees at both companies agreed that methods and processes are necessary in order to produce quality software. Especially in contexts where extreme reliability is needed, such as space exploration and telephone exchange software, developers saw strict adherence to methods as crucial. We can safely conclude that employing strict rules and regulations in the form of software processes and methods are viewed as essential in order to produce highly reliable software, both by software engineering researchers and practitioners

in the field. Much like the military operations studied by Snook, a software development project is set up to achieve a specified objective by following an initial set of global rules. It has a limited timeframe and a clearly defined "chain of command."

Having shown the relevance of Snook's theory, we know turn to some of these factors that contribute to the *drift* away from the processes and methods of the software development project as it is designed on the drawing board, focusing on four main issues: *pleasure and control, mental models, production pressures,* and *fragmentation of responsibility*.

3.5 Pleasures in Technology

Developers clearly value their skills and independence, and see too strict methods as boring and stifling creativity. As developer Knut says when talking about his experiences with previous employers:

> The consultancy business was rigid, very rigid. And I think that will sometimes be at the expense of enjoyment and creativity. You are in a way sitting like a robot in such a system, just sitting and producing something. That's a large part of the reason why I left the consultancy industry.[14]

Low and Woolgar found similar sentiments in their participant observation among software developers. One developer, William,

> spent a good deal of his time yawning and sitting slumped, staring into his computer screen … To him, the structure was overwhelming. The amount of written documentation that was required, the proliferation of structured documents set out in the CASE [Computer Aided Software Engineering] tool that he *had* to use, the deliberate sequence of work that he was supposed to follow. And all seemed to produce in him a defeated lethargy. (Low & Woolgar, 1993, p. 41, original emphasis).

---

[14] Interview K. M. Hansen, developer Telenor Mobile, Oslo: June 18, 2003.

Issues of pleasure and enjoyment seem to be a crucial factor for software developers. All the developers interviewed reported that feelings of pleasure and enjoyment were important motivating factors in their day-to-day work life. Most software developers see their job as an art as much as a craft, and the creative parts of the work, such as analysis and design, are especially valued. Some of the developers also reported taking enormous pleasure in the actual coding work; like Daniel, who has had periods of working very long hours because the job tasks have been "fun." When performing programming tasks he "can maintain an incredible work capacity without burning out."[15] This echoes the software professionals studied by Tine Kleif and Wendy Faulkner (2003). They found the same pleasures in software development and absorption in technical tasks. In that sense, the data gathered by interviewing software developers at Telenor Mobile and FIRM are remarkably similar to the results of Kleif and Faulkner's studies of hobbyist robot builders and professional software engineers. Creativity was mentioned by most of the developers as contributing to the fun and pleasure of software development. As Daniel puts it:

> What's fun is the creative part. It's either being architect, designing and working with concepts; or just programming, if you believe in what you are making.[16]

The pleasure seems to be derived from using one's technical skills in making something that works. Solving a problem in an especially efficient and elegant way is commonly associated with being a "hacker", a term denoting someone with an exceptional aptitude for programming and with immense pride and joy in his skills. The aesthetic dimension of programming seems to be important for a self-confessed hacker like Peter:

---

[15] Interview D. Bakkelund, developer Telenor Mobil, Oslo: June 16, 2003.
[16] Ibid.

> Q: Do you feel pleasure when programming?
> A: Yes! Yes, and that's connected to the hacker thing, that it feels good in your soul, in a way, if I'm able to do something that's beautiful. Absolutely.[17]

Or as Steinar puts it:

> I think the analysis and design part of the job is the most fun. Finding the solution to an intricate problem using software, doing it in an elegant way, preferably by using patterns … Putting together things you know from before, like structures, that is fun.[18]

Another recurring theme that crops up when asking the developers about their work, concerns issues of control and ownership. As Kleif and Faulkner suggest, technology in general and software in particular provides an arena for power and control and for overcoming uncertainties. The feeling of control seems to be a requisite for enjoying a task. As Steinar puts it when asked about the less enjoyable aspects of work:

> [W]hen you're *not in control* of the system you're supposed to work with … that's probably the least satisfactory of what I work with now, because you can be struggling with an error that turns out *not to be yours* for quite some time … That can be frustrating. [Emphasis added][19]

Rasmussen and Håpnes (1991) also found an astonishing fascination with computers and programming in their study of computer science students. An all-male group of "programming virtuosos" shunned classes and professors' assignments, choosing instead to sit all night at computer terminals experimenting and "work[ing] for the joy of the process and the grand feeling of achieving control" (p. 1111). Again there is an emphasis on the feeling of control as the main motivator and source of pleasure. However, among the developers at Telenor Mobile and FIRM, no evidence was found that indicated that the people

---

[17] Interview P. Myklebust, director of development FIRM, Oslo: June 23, 2003.
[18] Interview S. Lundeberg, developer Telenor Mobil, Oslo: June 13, 2003.
[19] Ibid.

taking pleasure in technology are people that feel less powerful and unable to cope with uncertainties in other areas of life, as Kleif and Faulkner seem to believe. None of the interviewees conformed to the stereotypical "nerd" image of the computer professional.

The developers clearly distinguish between tasks that are enjoyable or "fun" and those that are less so. As we have seen, tasks that allow developers to employ their creativity and exercise control are considered fun. Routine and repetitive tasks where the developer is not in total control are seen as boring or "not fun." Tasks that fall in the latter category are typically software testing, i.e. systematically exercising software modules or subsystems to ensure that they conform to requirements and do not fail or produce incorrect results. Estimating, i.e. trying to assess the time and resources needed to implement a certain task, is seen as an especially joyless task. This is probably due to the high level of uncertainty involved and the lack of control felt by the developer given the task of predicting future resource needs and timelines. Importantly, estimating and testing are recognized as the most crucial in increasing software quality and lowering vulnerability. Incorrect estimates are seen as the source of unrealistic deadlines and production pressures; and "testing is the only way of really getting rid of bugs in your software," as Daniel points out[20].

An important question is therefore whether the developers let the "fun factor" influence how well they perform these important tasks. Daniel admits that

> Developers are … not exactly immature and childish, but a bit like, you try all the time to adjust so you have fun. Most people working here … think programming is very fun, and so you try to sneak out of the things you don't think is fun.[21]

Writing proper technical documentation to aid in future maintenance of the software is also a crucial task that is seen as menial. Rodin feels that

---

[20] Interview D. Bakkelund, June 16, 2003.
[21] Ibid.

> Good developers are a special type of people and many of them hate [writing] documentation. Often it's the best [developers] that hate [writing] documentation, those who write the best code.[22]

While all the developers emphasize that they do not shirk their responsibilities even though some of their work tasks may feel less enjoyable, most agreed that the level of fun associated with a task will definitely influence how much work goes into it, and ultimately affect the quality of the software. The general feeling towards these tasks is a "get-it-over-with attitude," as Steinar calls it.[23]

It is worth noting that a majority of the developers were not trained in computer science or software engineering, but has strayed into the field from mathematics and other engineering disciplines, often after doing programming as part of the work within their original fields. This suggests that development work attracts people with specific personality traits and perhaps with a special aptitude for this kind of work. Whether this makes them especially prone to the factors discussed here, is an interesting avenue for further research.

3.6 Mental Models

While a fascination with the computer and a sense of joy in controlling it can be an asset for a software developer, helping him funnel his skills and creativity into his software, it can also be a source of pain. The sheltered environment of the university computer lab is rarely the norm in the professional work place. The software developer's intimate knowledge of the machine and its ordered world can be an obstacle when communicating with other groups involved in the development process. I propose that this is largely due to different *mental models*.

---

[22] Interview R. Lie, systems architect/developer Telenor Mobil, Oslo: June 20, 2003.
[23] Interview S. Lundeberg, June 13, 2003.

Paul Edwards (1996) suggests that much of the fascination felt by computing professionals stems from the simulated character of the "microworlds" they create and the omnipotent power and control the programmer can wield there (pp. 171-172). In order to build a computer program that is supposed to interact with human beings and assist them, the software developer will have to build a *mental model* of the relevant aspects of the real word. Due to the nature of computers and software, such a model will have to be based on the reductionist and deterministic principles that govern computers. The model must be "stripped of both social and emotional complexity" (Kleif & Faulkner, 2003, p. 313). This means that every aspect of the mental model will have to be spelled out and follow deterministically from the principles, rules and laws governing the microworld contained in the software system. No aspect of the software system can be left vague or open to interpretation. There is no latitude for social factors or human idiosyncrasies outside the mechanical workings of a positivist world. To put it bluntly, there is no room for constructivism inside a computer program. Regardless of whether one subscribes to theories of social constructivism or not, it does not require a stretch of the imagination to see that the intricacies of the real world tend to get lost when forced into a computer system; and the social complexities particularly so. Software developers, like other engineers, have been drilled in analytic and problem solving and tend to "perceive the world of mechanisms and machinery as embodying mathematical and physical principles alone" (Bocciarelli, 1994, as cited in Kleif & Faulkner, 2003, p. 313).

Gorman & Carlson (1990) use the concept of mental models when investigating the cognitive processes involved in technical innovation, using the development of the telephone by Alexander Graham Bell and Thomas Edison as case studies. They borrow the term from cognitive scientists who use it to describe "the models people have of themselves, others, the environment, and the things with which they interact." (Norman, 1988, as cited in Gorman &

Carlson, 1990, pp. 134-135). Of course inventors and software developers are not the only ones to construct a mental model of a given system:

> [T]hrough interaction with a target system, people formulate mental models of that system. These models need not be technically accurate … but they must be functional. A person, through interaction with the system, will continue to modify the mental model in order to get a workable result. Mental models will be constrained by various factors such as the user's technical background, previous experience with similar systems, and the structure of the human information processing system (Norman, 1983, pp. 7-8).

It is clear that different people with different background coming in contact with a computer system will have widely differing mental models of how the system works, what it is capable of doing, and what it takes to change it. The developers of a system possess intimate knowledge of the technical foundations and the assumptions behind it. Their managers, often without a technical background, see the system in terms of the time and resources needed to build it, as well as the requirements and constraints presented to them from the customer and other stakeholders having an interest in the system. The (present or future) users of the system form their mental models of it on the basis of their understanding of the real world domain, the user interface the systems presents them with, and the data input and output to and from the system. Most groups will have this "black box" view of the system, seeing it in terms of its external properties and behaviour. The system developers are of course the notable exception, with their mental models based on the inner workings of the system. This sets the stage for a variety of problems of communication between software developers, their managers and the users of the software that, as we shall see, profoundly affects the vulnerability of the software.

The concept of mental models bears striking resemblance to Bijker's (1995) notion of *technological frame*, capturing the idea that different social groups will have different "goals, key problems, problem-solving strategies (heuristics), requirements to be met by problem

solutions, current theories, tacit knowledge, testing procedures, and design methods and criteria" (p.123) relating to a specific artefact. Bijker also introduces the concept of degree of *inclusion* into a technological frame, describing "to what extent the actor's interactions are structured by that technological frame" (p. 143). This concept is important in analysing the apparent malleability or obduracy of a technological artefact, as it captures the degree of "sell-in" of actors in a specific technological frame or mental model. As Gorman and Carlson point out (p. 136), the technological frame does not include their concept of mechanical representation (p. 141), the actual physical objects with which the inventor (developer) or user interacts, using them in combination with abstract ideas to address new problems. In the current context, mechanical representations map to the software systems under construction or use, or other software systems with which the developer or user has had previous experience.

Within software engineering research, Peter Naur (1985) has proposed that programming can be understood mainly as "theory building," relating the software to its anticipated use. He suggests that programming "should be regarded as an activity by which programmers form or achieve a certain kind of insight, a theory, of the matters at hand" (p. 253). The main task of a software developer is therefore not the production of the program text itself, but the understanding of the aspects of the real world that are to be modelled, automated or replicated, and the formation of a theory of how this can be achieved by a computer program. Christiane Floyd (1992) expands on this and sees software development as a form of reality construction where developers rather than analysing requirements construct them from their own perspective, affected by their personal priorities and values. These views of software development emphasize the active part of the programmers in the construction of meanings of software systems, as well as the power they have to embed their view of reality and the social world in the artefacts they produce. All of these related concepts have one thing in common, namely that they embody the insight that different individuals will

have different ideas and notions about a software system and how it relates to them and their environment. In the following, the concept of mental models will be used as basis for the analysis.

Most of the developers interviewed expressed frustration with some of their project leaders and managers. At Telenor Mobile, most of the project leaders do not have a technical background, but have business or economy educations. At FIRM, with its development activities on a smaller scale and managers with technology backgrounds, this is less of a problem, even though the sentiments can be felt there as well. As Steinar has experienced, project leaders' lack of technical knowledge can be problematic, especially when eliciting requirements:

> I have seen on several occasions that they haven't quite been able to catch what is important because they are not technologists. So I believe that … there has to be technical personnel participating all the way from the start of any project, it's a problem if you don't.[24]

Otherwise, you risk "in the worst case [delivering] to the customer something they actually didn't ask for. That has occurred."[25]

Communicating with existing and potential users of their software can also be a source of frustration for a developer. Rodin puts it like this:

---

[24] Interview S. Lundeberg, June 13, 2003.
[25] Ibid.

> I have learned that it is incredibly difficult for a non-computer person to understand that it can be so damned difficult. They just can't comprehend that it can be so difficult, it's just a box with some stuff on the screen, it can't possibly be so difficult to do. They can't understand it. The times I have tried to work on this, I see that with a few exceptions it's often easier for me to get into their problem formulation than the other way around … For developers, users are a nuisance.[26]

Even communicating and cooperating with fellow developers can pose problems. Due to the complexity and intractability of software, developers who have not been directly involved in programming a specific piece of software will most often take a black box view of other developers' work.

At the bottom of these communicative problems are the different mental models that developers, managers, and users form of the system. It seems to be exceedingly difficult for people with different mental models of a system to find common ground and cooperate. Developers feel, perhaps justly, that their intimate knowledge of the system gives them a privileged insight. They seem to resist seeing the world from other people's viewpoint and ascribe the difficulties in communications to the lack of technical insights and interest in managers and users. In Bijker's (1995) parlance, their inclusion in their respective technological frames is too high for effective communication and cooperation. Each group struggles to make its technological frame or mental model the dominant one. This might be an especially painful struggle for the software developers. As Undheim (2002) found in his fieldwork among Telenor engineers, in the struggle between marketing people, managers and engineers, the engineers did not have the power or the "symbolic vocabulary to infect others" (p. 119), choosing instead to remain "silent". The engineers were left in their purely technical domain "wondering what is going on, and both parties ironise over the incompetence of the other" (p. 117).

---

[26] Interview R. Lie, June 20, 2003.

Both Telenor Mobile and FIRM employ people who specialise in bridging the gap between customers and users who provide the requirements and functional specifications for the software, and the software developers who are given the task of developing the system. These are people with the ability to see things from several groups' viewpoint simultaneously, moving from one technological frame to the other. Usually these "bridge builders" are developers, probably because the "technical" mental models or frames are difficult to enter without the knowledge and experience of a software developer. Rodin sees himself as a bridge builder:

> I think it's very difficult to realise what it's like not to understand [what it is like to be an ordinary user] … [It's] very difficult to realise how it is not to know something when you know it [yourself] … [Bridge builders] are people … who understand that the world is more than programming.[27]

Still, this gap between the developers and the non-technical stakeholders is probably the largest cause of vulnerable software. When customers are unable to formulate their requirements for a new system in such a way that developers can understand them and translate them into a form suitable for inclusion into a computer system, and developers are unable to communicate with the customers in order to understand how the software they are building will be used in a real world setting, the risk of ending up with software that is not used the way it is intended increases dramatically. As we saw in the case of the London Ambulance Service, this type of vulnerabilities can have grave consequences, as users struggle to make the software work in their world. The manner in which designers and engineers "inscribe" their ideas and notions into an artefact based on their assumptions about the potential users and their world has been described by Akrich (1993). She introduces the concept of "scripts" to indicate this idea. Software developers are thus able to "inscribe" the

---

[27] Ibid.

software systems with "scripts" representing their world views and ideas about the prospective users of the system.[28] Wyatt (1998) argues that the malleability and flexibility of the computer is illusory; in order for the technology to be used; "malleability has to be excluded during the process of development" (p. 6). The constraints on the users of a particular software system are therefore considerable and not easily changed.

3.7 Production Pressures

The software development team is usually supervised by a project leader whose job it is to establish the team, secure resources, plan the execution of the software development project, divide the project into subtasks, compile estimates, communicate with stakeholders, and monitor progress as the project moves along. As mentioned previously, many developers see project leaders' lack of technical skills as a source of frustration, causing problems of communication. We have also seen that the developers admit to giving the task of estimating a low priority, due to its perceived joylessness. A project leader without a technical background will have no independent basis for assuring the quality of the estimates he receives from the developers. Added to the fact that estimating software projects is notoriously difficult due to the large number of uncertainties involved, it is no wonder that correct estimates are something of a rarity in most software projects. Notoriously, estimates seem invariably to undershoot the actual effort needed for a specific task.

Incorrect estimates thus usually mean unrealistic deadlines. For project leaders and managers, a lot could be at stake by not keeping the forecasted deadlines. Customers may have been promised completion of a project within a specific date and may have based their own activities on this; in Telenor Mobile's case, a large marketing campaign for a new

---

[28] Although an actor-network theorist, Akrich emphasises the ideas and worldview of the designers being built into an artefact and adding to its obduracy (Hommels, 2001, p. 38; p. 169). The relevance to the present discussion is clear.

product may have been scheduled involving TV and newspaper commercials, making a missed deadline unbearable. All of this adds up to a tremendous pressure to keep deadlines and finish software modules within the estimated times. As Daniel says:

> I have often handed off software I felt could have used one more week of [work] … [The code] that you hand off is very chaotic, because of the time pressure you just about manage to cram in the functionality you need and then you don't have time to clean up afterwards to make it maintainable. It's like, just chaos.[29]

The consequences in terms of reliability and maintainability of the software should be obvious. Most of the developers feel that it is the testing of the software once it is coded that suffers the most under tight deadlines. Testing is the last task before a piece of software is regarded as completed, and is usually the place where the pressure of the deadline is felt most acutely. As Rodin says, when the developers are not given enough time to do testing properly, "What happens is that you test the best scenario or the things you know will work. Then you don't get to test all the failure scenarios and once [the program encounters a situation] outside the norm, it blows up."[30] Developer Per explains why this happens:

---

[29] Interview D. Bakkelund, June 16, 2003.
[30] Interview R. Lie, June 20, 2003.

> Q: Does it occur that deadlines are set that make the quality of [the software] suffer?
> A: Yes.
> Q: Why is that?
> A: Well, it's the business side who wants things out there, and who feels that everything is too slow and puts enormous pressure [on the developers], sells things before they're [done], sells things to a deadline that [is nonnegotiable].
> …
> Q: Do you feel that they lack understanding of the fact that it can take time because you have to maintain quality and security?
> A: Yes. Yes, absolutely.[31]

Again we see that the "silent engineers" (Undheim, 2002) are left to do their best within the confines of their technical domain. As Diane Vaughan found in her investigation into the Challenger space shuttle disaster (1996), a culture of institutionalized production pressure can be dangerous for an organization dealing with complex systems (pp. xii-xiv). At NASA, it played a major role in the tragic outcome of the shuttle launch; at Telenor Mobile or other software development organizations, it can lead to faulty software being put into production and the consequences that entails.

FIRM does not seem to suffer the same degree of production pressures. Their flexible approach to development processes, having a range of processes to choose from according to the scale of the project at hand, seems to yield beneficial results. Their experience with the different development processes, the ConfirmIT system and the application domain also enables them to produce more precise estimates, thus avoiding much of the production pressure. Although I am not able to draw any definite conclusions on the basis of the present material, I would expect that this leads to more reliable software, since the time spent on testing and quality assurance does not suffer to the same degree.

---

[31] Interview P. Hustad, systems architect/developer Telenor Mobile, Oslo: June 16, 2003.

3.8 Fragmentation of Responsibility

Although software testing is seen as crucial in order to ensure that the software is as bug-free as possible, both Telenor Mobile and FIRM employ additional strategies in order to achieve high quality software. A process of peer code review, where one developer's code is inspected by another developer is seen as a crucial stage in the software development process at both companies. However, the actual content of this stage seems to be unspecified and left to the individual developer performing this review, or QA (quality assurance) as it is called at Telenor Mobile. As with other tasks that do not directly produce code, it is at risk of being taken more lightly when the pressure is on to reach a tight deadline. Interestingly, the perception of the skill of the person whose work is up for review, versus the skill of the reviewer, can play a factor in determining how thorough the task is done, as Steinar volunteers:

> Q: Could it be that you take [QA] less seriously if you're in a hurry?
> A: That could happen. Or if you have a lot of confidence in the person who made the code. That's also a factor.
> Q: Right. So if you're assigned to do QA on the work of someone you consider being very skilled, it could be that you just assume that it works?
> A: Yes, that's a factor. I would probably do that if I was pressed for time, for instance, and … it was important to get this done, then I would be prone to go more lightly at it than if it was someone I didn't know … who made that code.[32]

Daniel confirms this; "It depends on who [wrote the code]. If it's someone I know is very good, then I might not go in depth, I would just look it over."[33]

Consequently, in a stressed situation the important QA steps could be taken very lightly, based on the perceived skill of a developer. It is not hard to imagine that a developer

---

[32] Interview S. Lundeberg, June 13, 2003.
[33] Interview D. Bakkelund, June 16, 2003.

who is reviewing code written by a developer with greater perceived skills will be hard pressed to raise a warning over code he does not understand, but simply assume that everything will work. This could in turn let serious errors slip through one of the few quality assurance measures, thus increasing the vulnerability of the software. It is also debateable whether errors and other shortcomings actually can be adequately identified simply by inspecting the textual program code of a piece of software.

Software bugs, i.e. logical flaws that result in program crash or faulty data, are always at the forefront of developers' attention, simply because their presence cannot be ignored. As discussed previously, bugs are not the only cause of vulnerabilities in software. Other issues concerning security and privacy are also important when assessing the vulnerability of a software system. However, among the developers interviewed, these issues seemed absent from their daily concerns. As Daniel puts it, "security is a neglected area within software development, mostly, and [there is] very little knowledge [about security issues] among developers. There's very little time and focus on it from management."[34] It can again be argued that the lack of technical skills and knowledge among project leaders and managers prevents them from recognizing the dangers of security and privacy flaws, much less implementing countermeasures. It is to a large extent left to the individual developer to ensure that such flaws do not exist in the software when it is put into use.

In much the same way that Snook found that important issues risk being neglected due to diffuse responsibility (2000, pp. 119-121), it seems clear that the majority of the developers feel that the responsibility for ensuring security and privacy lie elsewhere. Daniel says that concerns over security are not issues in his day-to-day work, because "there's a guy over there that's responsible for security, he knows it, the rest of us just use some stuff and then it will

---

[34] Ibid.

take care of itself most of the time."[35] Steinar concurs that since safety and privacy issues are handled by others "I don't have to think about such things. I'm sure there are others who have this as their area of focus, but I don't."[36]

Considering the pressures forcing developers to take shortcuts it can only be expected that security and privacy vulnerabilities will find their way into the finished products. These types of vulnerabilities are especially insidious, since they may not surface during the normal operation of a computer system. Unlike bugs and other errors which normally manifest themselves in obvious manners, security flaws can lie dormant for years before they are discovered and exploited.

3.9 The Results of Practical Drift

I have now identified several factors that can contribute to practical drift during the development of software, and established that software developers will drift away from the strict rules of the prescribed software development process. We saw that this could have a detrimental effect on the quality and reliability of the software, thus increasing the vulnerability of the software systems produced. Although neither Telenor Mobile nor FIRM have strict operational rules and procedures to the same extent as a military operation, they both have established software development processes and methods that are meant to ensure the reliability of the software. To see the relevance of Snook's theory in this context, we turn again to Figure 1 on page 33. The establishment of an actual software development project can be seen as a transition from quadrant 1 to quadrant 2, employing a baseline of rules and regulations. The social factors influencing the developers constitute a drift from quadrant 2 to quadrant 3. The software system under construction has typically been divided into modules

---

[35] Ibid.
[36] Interview S. Lundeberg, June 13, 2003.

and subsystems, each assigned to different developers or groups of developers. According to Snook's theory, each of these will have worked along with their tasks, imperceptibly drifting away from the baseline of process and methods; each of them, however, along different paths. Especially in situations where the development process feels overly restrictive, the developers will adapt to the demands of the practical tasks at hand, replacing the rule-based logics of action by task-based logics of action more suited to the environment the developers work in. As we have seen, the individual developers will make decisions to deviate from the rigour of strict methods based on what appears practical, but also according to the personal inclinations of the developers, emphasising those tasks that are associated with "fun." We also saw that production pressures, difference in mental models, and communication barriers prompted developers to deviate from following procedures designed to ensure software reliability.

Snook predicts a rapid shift from the relative stability of quadrant 3 when the situation suddenly changes from a loosely coupled one to a tightly coupled one, demanding that several subgroups work tightly together towards a common goal. In our present context, this would be the stage of the software project were modules and subsystems are brought together for systems and integration tests, and finally putting the complete system in operation. This is when different parts of the system developed by different parts of the project organization are supposed to start interacting and working together. As we learned above, the processes surrounding testing and subsystem integration are especially prone to drift due to social factors. The relevance of Snook's theory is clearly demonstrated, as it identifies this as a point where the risk of disaster is greatest and therefore demanding careful attention by developers and managers.

At Telenor Mobile and FIRM there were no stories of disasters on the same scale as the London Ambulance Service (LAS) case that started this thesis. Many of the developers could recount tales (mostly from previous employers) of potentially disastrous events only

averted by sheer luck or by working round the clock; or struggling to come to terms with hopelessly unrealistic deadlines and incompetent managers. More often than not, incidents like these are suppressed or forgotten and are never subjected to analysis within software development organizations. The fact that the LAS case was thoroughly investigated and analysed makes it nearly unique and provides an excellent opportunity to learn. We can find several of the social factors discussed here in the report from the official investigation into the LAS case (Page et al., 1993). The report describes how the project was supposed to use the PRINCE project management methodology, but "[a]lthough certain elements of the PRINCE methodology were used, *at least in the initial stages*, it was not used in a properly structured way through the duration of the project" (para. 3068, my emphasis). The report does not explicitly mention the conditions of the software developers whose task it was to build the CAD system, but it is clear that tremendous pressures were put on the project team to meet a nonnegotiable deadline for delivery of the finished system. The inquiry found that the company developing the software was "late in delivery of software and, largely because of the time pressures under which they were working, the quality of their software was often suspect" (para. 3079). The consequences were that tasks such as quality assurance (QA) and testing suffered (para. 3083-3086). Other important parts of the project methodology were disregarded, as software developers "in their eagerness to please users, often put through software changes 'on the fly' thus circumventing the official Project Issue Report (PIR) procedures whereby all such changes should be controlled … Such changes could, and did, introduce further bugs" (para. 3082). Communication between software developers and the Central Ambulance Control and other staff that were to use the CAD system was also flawed during design and implementation. Accordingly, "there was incomplete 'ownership' of the system by the majority of its users" (para. 1007o) leading to little participation from LAS staff and an expectation that the system would fail. More importantly, a lack of understanding of

the users' needs and requirements led to the design of a system with "a need for perfect input formation in an imperfect world" (para. 4007a). The deployment of a system that did not fulfil user needs also led to local adaptations on the part of the users:

> [S]atisfactory implementation of the system would require changes to a number of working practices. Senior management believed that implementation of the system would, in itself, bring about these changes. In fact many staff found it to be an operational 'strait jacket' within which they still tried to operate *local flexibility*. This caused further confusion within the system (para. 1007p, my emphasis).

In the LAS case, we are able to recognize many of the same social factors that were found among the developers at Telenor Mobile and FIRM. The factors mentioned here were by no means the only causes of the spectacular failure of the LAS CAD system, but they were certainly important in contributing to it. They provide us with a worst-case scenario of what practical drift can lead to in a software development organization.

We also saw that important differences existed between Telenor Mobile and FIRM, which had an influence on the vulnerability of the software produced at the respective companies. FIRM had a more flexible range of software development processes, thus being able to work with more realistic estimates and less production pressures. FIRM's developers also enjoyed a higher status within the company and more power to get their worldview across than what was the case at Telenor Mobile, where several layers of management exist between them and the top executives. There are strong indications that these differences in social organization influence the vulnerability of the software they develop.

From the users' perspective, inadequate communication with the software developers and conflicting mental models could lead to confusion about their requirements and to the development of software that does not address the needs of its users. As the LAS case so

brutally demonstrated, this could in turn expose further vulnerabilities both in the users and in the software itself.

4 Conclusion: Living with Vulnerability

4.1 Summary

We began this investigation at the London Ambulance Service and saw how a new computer system that was meant to make life easier for emergency staff and safer for the general population turned out to have the opposite effect. People's lives came at risk due to the vulnerabilities of a computer system, thus exposing the vulnerabilities of human beings in a technological society. In the LAS case, no grave technical flaws were found in the system itself; in a sense it only did what it was designed to do. This is something that this incident has in common with many accidents from a wide range of areas involving complex technical systems; technical malfunction may play a role, but it is usually social or organizational factors that turn minor incidents into fatal accidents. On the other hand, when accidents do *not* happen, or incidents are contained *before* they can escalate to damaging accidents, social and organisational factors usually play an important role as well, contributing to *less* risk.

The research question behind this thesis was to look at how social and organizational factors influence the vulnerability of software systems, placing this research within the field of science, technology, and society (STS) studies. Employing interdisciplinary approaches, scholars within this field investigate the role of social processes involved in shaping technologies and the societal and political implications of their development and use. While STS scholars traditionally have looked at technological artefacts during their inception and production, researchers investigating issues of risk and vulnerability have been more concerned with the hazards posed by new technologies. By examining the public's understanding of science and technology and their perceptions of the risks involved, they are able to gain insights into the ways our lives and social worlds are shaped by our coexistence

with risky technologies. In order to gain further understanding of the issues related to the research question at hand, I first looked at how the risks associated with modern industrial production have made scholars claim we live in a "risk society". We also saw how modern societies are immersed in technology and depend on its operation to the extent that we speak of our cultures as "technological cultures" and how the advent of information and communications technologies (ICTs) have fuelled the rise of an "information society" (chapter 2). Although researchers may disagree whether and to what extent these technologies represent something fundamentally new, it is no doubt in my mind that ICTs and the advent of global information networks have transformed how we conduct business, how we organize our workplaces, what kind of work we do, and how we communicate with others in ways that have profound social and political implications.

To further analyse the risks and vulnerabilities surrounding computer systems, we set out to investigate whether Charles Perrow's *normal accident theory* could be fruitfully applied to this avenue of research. His concepts of *complexity* and *coupling* were found to be directly relevant to computer systems and important tools in describing and analysing software and its vulnerability. Just as Perrow found in his analysis of nuclear power plants and chemical factories, faults are to be expected in most contemporary software applications. Such highly complex and tightly coupled software systems pose higher risks because of their inherent intractability and impenetrability exceeding human capacities for understanding. When employed in real life settings, the larger socio-technical system comprising the surrounding social organization as well as the technical system can attain even greater complexities that contribute to making accidents expected and indeed *normal*. However, scholars within the field of *high reliability* theory take a more optimistic view than Perrow, claiming that there are social and organizational measures that can be taken in order to improve the reliability of a system, thus lowering its vulnerability. While still acknowledging

the inevitability of accidents in complex systems, high reliability theorists feel that measures like continuous training and redundancy can and should be used to lower the risks associated with such systems. I concur with these sentiments, especially since Perrow's view is that some technologies like nuclear power plants are inherently too complex and risky, and that these technologies should be abandoned. However, more comprehensive research is certainly warranted into the role of social and organizational factors in high-risk, high-reliability systems.

While Perrow's theory proved useful as a starting point for assessing the risks and vulnerabilities involved in complex socio-technological systems, it lacks a convincing analysis of the social aspects of such systems. Specifically, it focuses solely on the inherent properties of systems, ignoring the fact that human beings and our complex social worlds are involved in high-risk systems. Perrow is more concerned with the static properties of complex systems and does not provide us with the tools needed for analysing the dynamic properties and individual and organizational issues that can contribute to vulnerabilities. His position is also a technological determinist one, seeing human and social interactions as unilaterally shaped by the properties of technological systems. For this reason, we turned to Scott A. Snook's theory of *practical drift*, combining ideas from normal accident theory and high reliability theories. Investigating the 1994 "friendly fire" incident over Northern Iraq where two U.S. Air Force fighter planes shot down two U.S. Army helicopters, Snook employs theories of behavioural psychology and organizational sociology to formulate the concept of *practical action*, where adaptations to global rules and regulations are done locally in parts of a larger organization for pragmatic and sensible reasons. These local changes mean that subgroups within an organization can drift away from the initial set of rules governing operations within the organization. This drift occurs differently in different subgroups, setting

the stage for disaster when conditions change and action has to be coordinated across multiple subgroups.

This theory of practical drift formed the framework for an empirical investigation among professional software developers in order to ascertain whether Snook's theory could be fruitfully employed in order to gain new understanding of the social and organizational issues affecting the production of software. I set out to identify factors that could be seen as contributing to practical drift in organizations developing software. My aim was to see how these factors could affect the vulnerability of the software itself, as well as the vulnerability of the users of this software. Software developers from two Norwegian companies, Telenor Mobile and FIRM, were interviewed about their attitudes towards risk and vulnerability, and their perception of factors that can affect the quality and vulnerability of the software the produce.

It can be debated whether the heavily designed and strictly regulated military organization that is the starting point for Snook's analysis can be said to be similar to a software development organization. This issue is important when assessing whether Snook's conclusions can be legitimately carried over to this new domain. The obvious differences in objectives should not distract us from the fact that there are important similarities in the way these two forms of organization operate, and that important lessons can be learned from Snook's analysis. A military operation and a software development project are similar in that they both are set up to achieve a clearly defined task in a specific timeframe. They are both governed by rules and guidelines derived from practice and experience from similar endeavours. Work processes, social organization and the "chain of command" are usually specified in detail before work starts and are tailored for the specific situation. In both military operations and software development projects, individuals are expected to show personal initiative in order to accomplish the task at hand, and in most cases successful completion of

the operation depends on this. In neither case do things go exactly as planned. Although there are differences in the goals and cultures of these two types of organization, I believe that the similarities are many and important enough to warrant further investigation.

When interviewing the software professionals at Telenor Mobile and FIRM, it became clear that they are prone to practical drift in much the same way as the military personnel in Snook's research. The developers see strict rules and processes as necessary in settings were extreme reliability is needed. However, too strict methods are seen as boring and stifling developer creativity. The individualistic nature of the developers will make them take practical, pragmatic action if they feel methods and processes are unnecessary or burdensome.

I was able to identify four areas where social factors turned out to be contributing to practical drift. Firstly, issues of *pleasure and enjoyment* seem to be an important factor for software developers, motivating them in their daily work and determining their priorities. By striving to spend as much time as possible with the more pleasurable tasks, important tasks such as estimating and testing run the risk of being taken less seriously. As these are the tasks designed for lowering risks and finding flaws in the software, this factor can contribute to increased vulnerability. Secondly, the different *mental models* employed by the different groups involved in software development affect their ability to communicate and achieve a common understanding of the goals and requirements of the system under development. This can cause software to be developed that does not fit well within its future use context, and could lead to vulnerabilities for the users of the software. As we saw in the case of the London Ambulance Service, a software system based on false assumptions about its users and their requirements failed horrifically as the emergency call operators and ambulance crews tried to make it do what they needed it to do. Thirdly, the *production pressures* imposed on developers by managers and project leaders to reach project deadlines can force them to take shortcuts and thus bypass quality assurance mechanisms prescribed by the development

processes. Fourthly, a *fragmentation of responsibility* can occur when the actual contents of quality assurance and other important processes are not clearly specified. Developers can take these tasks lightly if they feel that the perceived skill of a fellow developer makes them unnecessary, or if they feel that they are somebody else's responsibility entirely, as seems to be the case with security and privacy issues.

All these issues constitute social and organizational factors that contribute to the vulnerability of software systems and to the vulnerability of its users. This is by no means an exhaustive list, but it demonstrates some of the "non-technical" factors facing software developers, and it provides a novel starting point that could prove fruitful in the analysis of the vulnerabilities of software systems. Although these issues have been dealt with by researchers within the software engineering field, the addition of perspectives from social science theory in general and the STS field in particular can only improve the understanding of the complex social and organizational issues surrounding the practice of software development.

4.2 Implications

I have shown that software professionals are prone to practical drift in their daily practice of producing software. Processes and methods that are designed to ensure quality and reliable software are strayed from. This "drift" is understandable, since it has its basis in practical, pragmatic adjustments to local conditions and the tasks at hand. It is not due to incompetence or ill will. However, the result can clearly be vulnerable software, making its users vulnerable, and thereby making our societies vulnerable. It is therefore important that software professionals be aware of the potential consequences of vulnerabilities in their software. The links between practical adjustments and the vulnerabilities exposed to the users should be made clear. However, Snook warns us that we should not overshoot when trying to avoid

failures. Rather than tightening the rules, trying to define processes and methods that are aimed at producing error-free software, we should realise that the inherent complexities in computer systems make accidents normal. Systems should be designed in such away that failures are expected and dealt with. The LAS case is a grim reminder of what might happen if a system's design is based upon the near-perfect functioning of all parts of the system, including its human actors. As we learned from the software developers at Telenor Mobile and FIRM, the fun and pleasure of creative design and programming are big parts of what motivates them. It is not advisable to replace the space for individual initiative and autonomous work with bureaucratic rigour.

Developers, managers, and other groups involved in software production should also learn to recognize and respect each others "mental models". Each group should come to acknowledge the merits of the others' point of view and make the efforts necessary to gain an adequate understanding of their frames of mind. Managers and project leaders should strive to understand the technical issues that developers struggle with, and the unique skills of talented programmers. This would earn them the respect of their subordinates, and hopefully enable them to plan projects with more realistic resource allocations and deadlines, alleviating some of the production pressures that today result in vulnerable software. The developers should in turn learn to respect the social and organizational challenges tackled by their managers, who spend their days negotiating with demanding customers, unmotivated users, arrogant programmers, and impatient executives. All groups need to take the future users of the software more seriously. Since they are the ones that will be using the software, and ultimately suffer from its vulnerabilities, their view of the world needs to be taken as the basis for designing the system. Engineering-minded developers and manages should take into account the complexities of our social worlds and realize the futility of modelling all aspects of them in a finite digital computer. Design practices should to a greater extent incorporate the

dynamics and flexibility of a world of organizations and social configurations in constant change.

Security and privacy concerns seem to be a surprisingly neglected area within the two companies studied. This is probably symptomatic of a general attitude in most software development organizations. The number of challenges in producing working software is daunting even without these concerns. Nevertheless, in a world of viruses, worms, cracker attacks, and industrial espionage, software professionals need to take constant stock of their measures against these threats. Our increased vulnerability as data about all aspects of our lives are stored and processed by computer should be taken more seriously.

Social science theory has long been ignored by most researchers and practitioners of software engineering. At best, theories and methods from ethnography, behavioural science and other fields have been seen as tools to be used to solve software engineering problems, for instance for studying future software users and eliciting requirements from them. By instead using these tools for studying the software engineers themselves and their practices, important insights can be gained that will help us better understand the foundations of software engineering, its assumptions about the world and social organization and interaction, and software's role in home and work life. These are results that will benefit both software engineering and the social study of technology. The STS field is especially well equipped to conduct this research, and I hope that this thesis has succeeded in pointing out some areas that could benefit from such a perspective. In the next section I try to identify a few avenues for further inquiry.

4.3 Directions for Future Research

Snook's theory on practical drift is important in that it identifies the concepts of practical action and the drift away from established rules that it can entail. As we have seen, these

concepts can also be applied within research on software development. However, it seems to assume that things would be less risky if rules and regulations are followed. Clearly, practical drift does not occur through malice or incompetence on the part of the individuals within the organization, but by rational and pragmatic decisions taken by people doing their best. This is maybe the biggest weakness of Snook's theory; it does not clearly prescribe an alternative that would counteract practical drift or incorporate it as a factor when designing organizations and processes. Being aware of practical drift and anticipating it is a step in the right direction, but more research is needed to learn more about how we can take these social factors into account and lower the risks and vulnerabilities associated with our technologies. The path forward lies in opening up the technologies and the social processes surrounding them. Having the anticipation of failures and breakdowns built into systems is in my opinion the key to future low-risk systems. STS scholars should be able to make a significant contribution here, with their insights into the nature of science and technology, and their body of research into how new technologies are formed.

Another weak point in Snook's theory is his underlying assumption that practical drift is unequivocally negative. He acknowledges the fact that the local adaptations constituting drift are all practical from a local perspective. He even emphasizes that a strictly rule-based logic of action can be directly counterproductive when it comes to achieving the tasks at hand. However, he does not draw the conclusion that some of these local optimizations might be to the benefit of the system in question. While my empirical material does not support any definite conclusions, it seems to me that the software developers' resistance towards overly strict methods ("overkill" as they call it) may sometimes be the healthy reaction of skilled professionals. How to distinguish between a benign course alteration and a dangerous case of practical drift is an entirely different matter, however.

Investigating whether and in what cases practical action can lower the vulnerability of a system, is an interesting avenue for future research.

When it comes to studying software and software developers, an important alternative to the standard model of software development has proven increasingly successful the past decade. The so-called *free/libre* or *open source* software community has pioneered a development model based on free access to software, freely available source code and voluntary contribution from developers (DiBona et al., 1999; Moody, 2002; Raymond, 2001; Tuomi, 2001). This seemingly unviable conflation of hacker culture and anti-capitalist idealism has managed to produce software with purportedly equal or higher quality (in terms of reliability and number of bugs and security flaws) than the usual proprietary industry model. Some research has been conducted into this phenomenon, but in the context of drift and vulnerability future research could uncover whether open source developers are less prone to practical drift than industry developers, and how the differences in development approaches affect the vulnerability of the software. This research could start with the notions of pleasure, control, and power explored here. Developers within the open source community contribute to projects as they see fit and according to personal skills and interests, thereby benefiting from the effects of having developers who derive pleasure from the programming tasks and are motivated by doing what they find to be "fun" at any given time. The rewards for the developers are in terms of respect and admiration from their peers rather than financial gains. Combined with a review process where the program code is continuously open to scrutiny and improvement by every other developer, the open model should at first glance guarantee better software. However, the geographical distribution of developers, lack of central authority to ensure that the routine tasks are done as well as the "fun" tasks, and potential high turnover of developers carry with them their own set of risks and vulnerabilities that can spill over into the software.

Outsourcing parts of the software development process to low-cost developers in third-world countries such as India are being increasingly explored as a way of reducing the costs associated with the production of software (see for instance Kobitzsch et al., 2001). More research, for instance in the form of case studies of software development projects where a significant portion of the development work is done in a remote location, should provide a better understanding of the social and cultural issues involved. The communication of mental models needed for successful software development across cultural barriers should be studied and the implications for software vulnerability analysed. It is my firm belief that such research would conclude that the risks involved in this type of outsourcing far outweigh any potential benefits. The difficulties of communicating mental models and cooperating on building software are daunting even when developers, managers and users are located in the same building. Software development spanning continents, time zones, languages and cultures would in my opinion be fraught with social and cultural issues that would make any such venture highly risky. *Offshore outsourcing*, as it is sometimes called, could conceivably work in limited, technical domains were the requirements can be unambiguously stated in advance, and where the required interaction with future users is at a minimum.

Most of the research on drift and vulnerability has focused on accidents and disasters, usually of the spectacular kind, such as nuclear accidents and friendly fire incidents. Another avenue of research which could prove rewarding, would be to study successful organizations that are able to deliver software on time, on budget, and with the correct functionality. By comparing their social and organizational contexts with those of less successful counterparts, important insights could be acquired that would benefit the field of software engineering.

While this thesis has had the development of software as its focus, and dealt primarily with software developers, we touched somewhat on the role of the other groups involved in the production and use of software systems, especially the end-users. They are the individuals

who are expected to use the software after it is developed and put into production. A lot of research has gone into the design of user interfaces and to make computer systems easier to use. Similarly, software engineering research has taken a particular interest in requirements elicitation from the end-users. Less research, however, has been conducted on exactly how misunderstood or unstable requirements contribute to vulnerable software during the development process; how mismatches between end-user requirements and actual system operation can increase the vulnerability of the users themselves; and how users cope with vulnerable software. These are all avenues of research that can benefit from a closer cooperation between researchers from software engineering and the social sciences, especially the STS field.

4.4 Anticipating Vulnerability

In one of the most cited and debated articles within the software engineering field, Frederick Brooks claimed that the *essence* of software engineering was such that there would probably not be a "single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity [of software projects]" (Brooks, 1987, p. 10) within the next decade. Using images from popular mythology, Brooks compared such a development to the "silver bullet" needed to slay the "werewolf" of inherent complexity. 16 years later, most of his critics will have to concede that although a lot of new developments within software engineering were each heralded as an answer, none has proven to be the decisive silver bullet. Nothing on the horizon indicates that a werewolf killer will be found in the near future. This echoes Perrow's view about our limited ability to understand complex systems due to their intrinsic properties. The weak points of Perrow's theory notwithstanding; this is a phenomenon we have to come to grips with as we surround ourselves with more computers and software, each linked to

other, complex systems. We are increasingly basing our existence on the faultless, continuous operation of these networks.

To me, the way forward lies in accepting these risks, shedding our naïve faith in scientific and technological solutions that are supposed to make complex systems risk-free. Clearly, the social and organizational issues touched upon in this thesis, as well as the properties of complex technological systems, show us that we will have to learn to live with risks and vulnerabilities. This is especially evident within the field of ICTs, as any computer user can attest to. Efforts should therefore be directed to better understanding the complex intertwining of the social and the technical that influences the vulnerabilities of our technologies and thereby of our societies and our selves. Theories from the STS field, with their emphasis on studying both the social and the technical, should prove invaluable in gaining understanding of how we can build software systems and other technologies in a way that takes risks and vulnerabilities into account. Software developers can be trained to recognize the social nature of their work, rather than seeing it as a purely technical task. Realizing that other groups may have other mental models or technological frames of their understanding of software systems, can help developers communicate better with users and other stakeholders, thereby producing software that is better suited to its future use. By better understanding software developers, their motivations, and social world, new processes can be created that makes software development more predictable to managers, eliminating production pressures as a source of software vulnerabilities. With this new awareness, software developers could focus on organizing their work in new ways, producing software that is made less vulnerable not by *eliminating* risks, but by *anticipating* them.

Appendix A:List of interviewees

Telenor Mobile

Name	**Position**	**Date of interview**
Paul Skrede	Section Manager	13.06.2003
Steinar Lundeberg	Developer	13.06.2003
Daniel Bakkelund	Architect/Developer	16.06.2003
Per Hustad	Developer	16.06.2003
Knut Marius Hansen	Configuration Manager/Developer	18.06.2003
Stian Dahle	Developer	19.06.2003
Rodin Lie	Architect/Developer	20.06.2003

FIRM

Name	**Position**	**Date of interview**
Peter Myklebust	Director of Development	23.06.2003
Trond Johansen	QA Manager	23.06.2003
Hans Olav Damskog	Developer	25.06.2003
Øyvind Forsbak	Developer	26.06.2003
Kjell Tore Hveding	Developer	26.06.2003

References

Abbate, J. (1999). Cold war and white heat: The origins and meanings of packet switching. In D. MacKenzie & J. Wajcman (Eds.), *The social shaping of technology* (2nd ed.) (pp. 351-371). Buckingham: Open University Press.

Acid rain and critical loads (2001). *Electricity Association Environmental Briefing, 33.* Retrieved September 19, 2003, from http://www.energy.org.uk/Activtys/EnvBrief/AcidRain.pdf

Akrich, M. (1993). The de-scription of technical objects. In W. E. Bijker & J. Law (Eds.), *Shaping technology/building society: Studies in sociotechnical change* (pp. 205-224). Cambridge, MA: MIT Press.

Beck, U. (1992). *Risk society: Towards a new modernity* (M. Ritter, Trans.). London: Sage. (Original work published 1986)

Bijker, W. E. (1995). *Of bicycles, bakelites, and bulbs: Towards a theory of sociotechnical change.* Cambridge, MA: MIT Press

Bijker, W. E. (2001). Understanding technological culture through a constructivist view of science, technology, and society. In S. H. Cutcliffe & C. Mitcham (Eds.), *Visions of STS: Counterpoints in science, technology, and society studies* (pp. 19-34). Albany, NY: State University of New York Press.

Blaikie, P., Cannon, T., Davis, I., & Wisner, B. (1994). *At risk: Natural hazards, people's vulnerability, and disasters.* London: Routledge.

Bocciarelli, L. L. (1994). *Designing engineers.* Cambridge, MA: MIT Press.

Bowden, G. (1995). Coming of age in STS: Some methodological musings. In S. Jasanoff, G. E. Markle, J. C. Petersen & T. Pinch (Eds.), *Handbook of science and technology studies* (pp. 64-79). London: Sage.

Brady, T., Tierney, M., & Williams, R. (1992). The commodification of industry applications software. *Industrial and Corporate Change 1* (3), 489-514.

Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer, 20* (4), 10-19.

Castells, M. (1996-8). *The information age: Economy, society and culture* (Vols. 1-3). Oxford: Blackwell.

Ceruzzi, P. (1999). Inventing personal computing. In In D. MacKenzie & J. Wajcman (Eds.), *The social shaping of technology* (2nd ed.) (pp. 64-86). Buckingham: Open University Press.

Collins, H. M. (1984). Researching spoonbending: Concepts and practice of participatory fieldwork. In C. Bell & H. Roberts (Eds.), *Social researching: Politics, problem, practice* (pp. 54-69). London: Routledge & Kegan Paul.

Collins, H. M. (1995). Science studies and machine intelligence. In S. Jasanoff, G. E. Markle, J. C. Petersen & T. Pinch (Eds.), *Handbook of science and technology studies* (pp. 286-301). London: Sage.

DiBona, C., Ockman, S., & Stone, M. (Eds.). (1999). *Open sources: Voices from the open source revolution.* Sebastopol, CA: O'Reilly.

Dittrich, Y., Floyd, C, & Klischewski, R. (Eds.). (2002). *Social thinking – software practice.* Cambridge, MA: MIT Press.

Edwards, P.N. (1995). From "impact" to social process: Computers in society and culture. In S. Jasanoff, G. E. Markle, J. C Petersen & T. Pinch (Eds.), *Handbook of science and technology studies* (pp. 257-285). London: Sage.

Edwards, P. N. (1996). *The closed world: Computers and the politics of discourse in cold war America.* Cambridge, MA: MIT Press.

Einarsson, S., & Rausand, M. (1998). An approach to vulnerability analysis of complex industrial systems. *Risk Analysis, 18* (5), 535-546.

Flowers, S. (1996). *Software failure - management failure: Amazing stories and cautionary tales.* Chichester: John Wiley & Sons.

Floyd, C. (1992). Software development as reality construction. In C. Floyd, H. Züllinghoven, R. Budde & R. Keil-Slawik (Eds.), *Software development and reality construction* (pp. 86-100). Berlin: Springer-Verlag.

Floyd, C., Züllinghoven, H., Budde, R., & Keil-Slawik, R. (Eds.). (1992). *Software development and reality construction.* Berlin: Springer-Verlag.

Gorman, M. E., & Carlson, W. B. (1990). Interpreting invention as a cognitive process: The case of Alexander Graham Bell, Thomas Edison, and the telephone. *Science, Technology, & Human Values, 15* (2), 131-164.

Green, E., Owen, J., & Pain, D. (Eds.). (1993). *Gendered by design? Information technology and office systems.* London: Taylor & Francis.

Hammersley, M., & Atkinson, P. (1995). *Ethnography: Principles in practice* (2nd ed.). London: Routledge.

Hanks, P. (Ed.). (1986). *Collins English Dictionary* (2nd ed.). London: Collins.

Hommels, A. (2001). *Unbuilding cities: Obduracy in urban sociotechnical change* (Doctoral dissertation). Maastricht: Universitaire Pers Maastricht.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process.* Reading, MA: Addison-Wesley.

Jasanoff, S. (1994). Introduction: Learning from disaster. In S. Jasanoff (Ed.), *Learning from disaster: Risk management after Bhopal* (pp. 1-21). Philadelphia: University of Philadelphia Press.

Kleif, T., & Faulkner, W. (2003). "I'm no athlete [but] I can cake this thing dance!" – Men's pleasures in technology. *Science, Technology, & Human Values, 28* (2), 296-325.

Klischewski, R., Floyd, C., & Dittrich, Y. (2002). Introduction. In Y. Dittrich, C. Floyd & R. Klischewski (Eds.), *Social thinking – software practice* (pp. ix-xii). Cambridge, MA: MIT Press.

Kobitzsch, W., Rombach, D., & Feldmann, R. L. (2001). Outsourcing in India. *IEEE Software, 18* (2), 78-86.

Latour, B., & Woolgar, S. (1986). *Laboratory life: The construction of scientific facts.* Princeton, NJ: Princeton University Press. (Original work published 1979)

Leveson, N. G. (1995). *Safeware: System safety and computers.* Reading, MA: Addison-Wesley.

Leveson, N. G., & Turner, C. S. (1993). An investigation of the Therac-25 accidents. *IEEE Computer 26* (7), 18-41.

Library of failed information systems projects (n. d.). University of Wolverhampton, School of Computing and Information Technology. Retrieved May 7, 2003, from http://www.scit.wlv.ac.uk/~cm1995/cbr/library.html

Low, J., & Woolgar, S. (1993). Managing the social-technical divide: Some aspects of the discursive structure of information systems development. In P. Quintas (Ed.), *Social dimensions of systems engineering: People, processes, policies and software development* (pp. 34-58). London: Ellis Horwood.

MacKenzie, D. (1993). Negotiating arithmetic, constructing proof: The sociology of mathematics and information technology. *Social Studies of Science 23* (1), 37-65.

Mackenzie, D., & Wajcman, J. (1995). *The social shaping of technology* (2nd ed.). Buckingham: Open University Press.

Martiniussen, E. (2002, May 21). Blair will consider cleaning of Technetium-99. *Bellona Web.* Retrieved September 19, 2003, from http://www.bellona.no/en/energy/nuclear/sellafield/24281.html

Moody, G. (2002). *Rebel code: Linux and the open source revolution.* London: Penguin.

Murray, F., & Knights, D. (1990). Competition and control: The strategic use of IT in a life insurance company. In K. Legge, C. W. Clegg & N. J. Kemp (Eds.), *Case studies in information technology.* Oxford: Blackwell.

Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming, 15,* 253-261.

Neumann, P.G. (1995). *Computer-related risks.* New York: ACM Press.

Newkirk, J., & Martin, R. C. (2001). *Extreme Programming in practice.* Reading, MA: Addison-Wesley.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & R. Stevens (Eds.), *Mental models* (pp. 7-15). Hillsdale, NJ: Lawrence Erlbaum.

Norman, D. A. (1988). *The psychology of everyday things.* New York: Basic Books.

Page, D., Williams, P., & Boyd, D. (1993, February). *Report of the inquiry into the London Ambulance Service,* South West Thames Regional Health Authority. Electronic version retrieved May 14, 2003, from http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las/lascase0.9.pdf

Perrow, C. (1999). *Normal accidents: Living with high-risk technologies* (2nd ed.). New York: Basic Books. (Original work published 1984)

Pressmann, R. S., & Ince, D. (2000). *Software engineering: A practitioner's approach.* (European adaptation, 5th ed.). London: McGraw-Hill.

Quintas, P. (Ed.). (1993). *Social dimensions of systems engineering: People, processes, policies and software development.* London: Ellis Horwood.

Rasmussen, B., & Håpnes, T. (1991). Excluding women from the technologies of the future? *Futures, 23* (10), 1107-1119.

Raymond, E. S. (2001). *The cathedral & the bazaar: Musings on Linux and open source by an accidental revolutionary* (Revised ed.). Sebastopol, CA: O'Reilly.

Renn, O. (1992). Concepts of risk: A classification. In S. Krimsky & D. Golding (Eds.), *Social theories of risk* (pp. 53-79). Westport, CT: Praeger.

Roberts, K. (1990). Some characteristics of one type of high reliability organization. *Organization Science, 1* (2), 160-176.

Rönkkö, K., & Lindeberg, O. (2000). 'Bad practice' or 'bad methods': Software engineering and ethnographic perspectives on software development. *Proceedings of the IRIS, 23*. Trollhättan: University of Trollhättan/Uddevalla.

Sagan, S. D. (1993). *The limits of safety: organizations, accidents and nuclear weapons.* Princeton, NJ: Princeton University Press.

Seale, C. (1998). Qualitative interviewing. In C. Seale (Ed.), *Researching society and culture* (pp. 202-216). London: Sage.

Snook, S. A. (2000). *Friendly fire: The accidental shootdown of U.S. Black Hawks over Northern Iraq.* Princeton: Princeton University Press.

Stake, R. E. (1994). Case Studies. In N. K. Denzin & Y. S. Lincoln (Eds.), *Handbook of qualitative research* (pp. 236-247). London: Sage.

Tuomi, I. (2001). Internet, innovation, and open source: Actors in the network. *First Monday*, *6* (1). Retrieved from http://firstmonday.org/issues/issue6_1/tuomi/index.html

Undheim, T. A. (2002). Visionary managers and silent engineers. In T. A. Undheim, *What the Net can't do: The everyday practice of Internet, globalization, and mobility* (Doctoral dissertation) (pp. 93-124). Trondheim: Department of Sociology and Political Science, Norwegian University of Science and Technology, Faculty of Social Sciences and Technology Management.

Van Loon, J. (2000). Virtual risks in an age of cybernetic reproduction. In B. Adam, U. Beck & J. Van Loon (Eds.), *The risk society and beyond: Critical issues for social theory* (pp. 165-182). London: Sage.

Van Loon, J. (2002). *Risk and technological culture: Towards a sociology of virulence.* London: Routledge.

Vaughan, D. (1997). *The Challenger launch decision: Risky technology, culture, and deviance at NASA.* Chicago: University of Chicago Press.

Wackers, G. (n. d.). *Notions of vulnerability.* Unpublished manuscript.

Warner, F. (1992). Introduction. In *Risk: Analysis, perception and management.* London: The Royal Society.

Webster, F. (2002). *Theories of the information society* (2nd ed.). London: Routledge

Williams, R., & Edge, D. (1996). The social shaping of technology. *Research Policy, 25,* 865-899.

Women in computing: Attracting more women to studies in computer science and communication technology at the Norwegian University of Science and Technology (NTNU) (n.d.). Retrieved August 7, 2003 from the Norwegian University of Science and Technology web site: http://datajenter.ntnu.no/2002_v2/english.html

Wyatt, S. (1998). *Technology's Arrow: Developing information networks for public administration in Britain and the United States.* Maastricht: Universitaire Pers Maastricht.